

**СОФИЙСКИ УНИВЕРСИТЕТ СВ. КЛИМЕНТ ОХРИДСКИ  
ФАКУЛТЕТ ПО МАТЕМАТИКА И ИНФОРМАТИКА**

**ТЕМИ ОТ КУРСА ПО  
Изкуствен Интелект**

---

**РАЗВИТИ ТЕМИ ЗА КУРСА ПО „Изкуствен Интелект“ ЧЕТЕН от ИВАН КОЙЧЕВ**

# Съдържание

<b>1 ПРЕДМЕТ И ЦЕЛИ НА ИЗКУСТВЕНИЯТ ИНТЕЛЕКТ. ОСНОВНИ НАПРАВЛЕНИЯ.....</b>	<b>9</b>
1.1 Какво е ИИ?.....	9
1.1.1 Да действаш като човек: тест на Тюринг.....	10
1.1.2 Да мислиш като човек: когнитивен (познавателен) подход за моделиране.....	11
1.1.3 Да мислим рационално: „закони на мисълта“.....	12
1.1.4 Да действаш рационално: подход на рационалния агент.....	13
1.2 Основи на изкуствения интелект.....	13
1.2.1 Философия.....	13
1.2.2 Математика .....	14
1.2.3 Икономика.....	16
1.2.4 Неврология .....	16
1.2.5 Психология.....	17
1.2.6 Компютърно инженерство.....	18
1.2.7 Теория на управлението и кибернетиката .....	18
1.2.8 Лингвистика.....	19
1.3 Основни направления .....	19
1.4 Ресурси.....	20
<b>2 ИНТЕЛИГЕНТНИ АГЕНТИ.....</b>	<b>21</b>
2.1 Обобщение .....	21
2.2 Речник .....	22
2.3 Ресурси.....	24
<b>3 ПРОСТРАНСТВО НА СЪСТОЯНИЯТА – ОСНОВНИ ПОНЯТИЯ И ЗАДАЧИ.....</b>	<b>25</b>
3.1 Основни понятия.....	25
3.1.1 Основни дефиниции .....	25
3.1.2 Представяне на пространството на състоянията .....	26
3.2 Действия, свързани с пространството на състоянията .....	26
3.2.1 Генериране на състояния .....	26
3.2.2 Оценка на състояние .....	26
3.2.3 Дефиниране на цялото пространство на състоянията .....	26
3.2.4 Основни типове задачи, свързани с пространството на състоянията .....	26
3.3 Основни задачи .....	27
3.3.1 Агент, основан на решаването на задача .....	27
3.3.2 Формулиране на целта.....	27
3.3.3 Формулиране на проблема .....	27
3.3.4 Търсене на решение .....	28
3.4 Добре дефинирани задачи и решения .....	29
3.4.1 Начално състояние.....	29
3.4.2 Възможни действия.....	29
3.4.3 Оценка за целево състояние .....	29
3.4.4 Цена на пътя.....	29
3.4.5 Формулиране на задача .....	30
3.5 Примерни задачи .....	31
3.5.1 Учебни задачи.....	31

3.5.2 <i>Проблеми от реалния свят</i> .....	33
3.6 Търсене на решения .....	34
3.7 Резюме .....	36
3.8 Речник .....	36
3.9 Ресурси.....	36
<b>4 МЕТОДИ НА 'СЛЯПО' ТЪРСЕНЕ НА ПЪТ ДО ОПРЕДЕЛЕНА ЦЕЛ .....</b>	<b>38</b>
4.1 Обща постановка.....	39
4.2 Търсене в широчина (BREADTH-FIRST SEARCH) .....	39
4.3 Търсене с непроменяща се цена (UNIFORM-COST SEARCH).....	42
4.4 Търсене в дълбочина (DEPTH-FIRST SEARCH) .....	44
4.5 Ограничено търсене в дълбочина (DEPTH-LIMITED SEARCH) .....	46
4.6 Итеративно търсене в дълбочина (ITERATIVE DEEPENING DEPTH-FIRST SEARCH ).....	47
4.7 Двупосочното търсене (BIDIRECTIONAL SEARCH).....	49
4.8 Заключение .....	50
4.9 Методи на 'сляпо' търсене на път до определена цел   Речник .....	50
4.10 Ресурси.....	51
<b>5 МЕТОДИ НА ЕВРИСТИЧНОТО ТЪРСЕНЕ НА ПЪТ ДО ОПРЕДЕЛЕНА ЦЕЛ .....</b>	<b>52</b>
5.1 Какво е евристично търсене? .....	52
5.1.1 <i>Обща характеристика</i> .....	53
5.1.2 <i>Критерии за използване на евристични функции</i> .....	53
5.2 Метод на най-доброто спускане (BEST-FIRST SEARCH) .....	54
5.3 Лакомо търсене (GREEDY BEST-FIRST SEARCH).....	55
5.4 Търсене с минимизиране на общата цена на пътя (A*).....	57
5.5 Евристично търсене с ограничаване на паметта (MEMORY-BOUNDED HEURISTIC SEARCH).....	60
5.6 Рекурсивен метод на най-доброто спускане (RECURSIVE BEST-FIRST SEARCH) .....	61
5.7 Опростен евристичен алгоритъм с ограничаване на паметта (SIMPLIFIED MEMORY-BOUNDED A*) .....	63
5.8 Евристични функции .....	64
5.9 Хипотеза на евристичното търсене (HEURISTIC SEARCH HYPOTHESIS) .....	65
5.10 Речник .....	65
5.11 Ресурси.....	66
<b>6 ЛОКАЛНО ТЪРСЕЦИ АЛГОРИТМИ. ГЕНЕТИЧНИ АЛГОРИТМИ .....</b>	<b>67</b>
6.1 Какво представляват локално търсещите алгоритми ? .....	68
6.2 Катерене по хълм (HILL-CLIMBING) .....	68
6.3 Рандомизирани алгоритми (RANDOMIZED). .....	71
6.3.1 <i>Случайна разходка (Random walk)</i> . .....	71
6.3.2 <i>Симулирано каляване (CK) (Simulated annealing)</i> . .....	71
6.3.3 <i>Генетични алгоритми</i> .....	73
6.4 Речник .....	77
6.5 Ресурси.....	77
<b>7 УДОВЛЕТВОРЯВАНЕ НА ОГРАНИЧЕНИЯТА.....</b>	<b>78</b>
7.1 Речник .....	78
7.2 Формулировка .....	79
7.2.1 <i>Пример: Оцветяване на карта</i> .....	79
7.2.2 <i>Граф на ограниченията</i> .....	81
7.2.3 <i>Инкрементална формулировка</i> .....	81
7.2.4 <i>Типове променливи</i> .....	81

7.2.5 <i>Типове ограничения</i> .....	82
7.3 Търсене с възврат .....	83
7.3.1 <i>Ред на променливите и стойностите</i> .....	84
7.3.2 <i>Разпространяване на информацията чрез ограничения</i> .....	85
7.3.3 <i>Интелигентно търсене с възврат</i> .....	87
7.4 Локално търсещи алгоритми за ЗУО.....	88
7.5 Структура на проблема .....	89
7.6 Литература.....	90
<b>8 ИЗБОР НА СТРАТЕГИЯ ПРИ ИГРА ЗА ДВАМА ИГРАЧИ - МИНИМАКСНА ПРОЦЕДУРА И АЛФА-БЕТА ОТСИЧАНЕ.....</b>	<b>91</b>
8.1 Увод.....	91
8.2 Минимаксна процедура.....	92
8.3 А-Б отсичане .....	94
8.4 Речник.....	96
8.5 Литература.....	96
<b>9 ПРЕДСТАВЯНЕ И ИЗПОЛЗВАНЕ НА ЗНАНИЯ - ОСНОВНИ ПОНЯТИЯ И ФОРМАЛИЗМИ.....</b>	<b>97</b>
9.1 Какво е знание? .....	98
9.1.1 <i>Разлики между данни и знания</i> .....	98
9.1.2 <i>Типове знания</i> .....	98
9.2 Какво е представяне и използване на знания? .....	98
9.2.1 <i>Разлики между бази от данни и бази от знания</i> .....	99
9.2.2 <i>Хипотеза за представянето на знания</i> .....	99
9.3 Основни формализми за ПИЗ .....	100
9.3.1 <i>Аспекти на формализмите за ПИЗ</i> .....	100
9.3.2 <i>Основни типове формализми за ПИЗ</i> .....	100
9.3.3 <i>Основни формализми за ПИЗ</i> .....	100
9.4 Логическо представяне на знания.....	100
9.4.1 <i>Какво е логика?</i> .....	100
9.4.2 <i>Съждителна логика</i> .....	101
9.4.3 <i>Логика от първи ред</i> .....	101
9.4.4 <i>Логики от по-висок ред</i> .....	102
9.4.5 <i>Други логики</i> .....	102
9.4.6 <i>Предимства и недостатъци на логическите системи</i> .....	103
9.5 Семантични мрежи .....	103
9.5.1 <i>Използване на знания, представени чрез семантични мрежи</i> .....	104
9.5.2 <i>Понятийни графи</i> .....	105
9.5.3 <i>Предимства и недостатъци на семантичните мрежи</i> .....	105
9.6 Продукционни правила.....	106
9.6.1 <i>Системи, основани на правила</i> .....	106
9.6.2 <i>Пример на система от продукционни правила</i> .....	107
9.6.3 <i>Предимства и недостатъци на продукционните правила</i> .....	107
9.7 Фреймове .....	108
9.7.1 <i>Представяне на знания чрез фреймове</i> .....	108
9.7.2 <i>Предимства и недостатъци на фреймовете</i> .....	109
9.8 Характеристики на добрия формализъм за ПИЗ .....	109
9.9 Речник.....	110
9.10 Ресурси.....	110

<b>10 ПЛАНИРАНЕ НА ДЕЙСТВИЯТА. ЧАСТИЧНО НАРЕДЕНИ ПЛНОВЕ.....</b>	<b>111</b>
10.1    Дефиниция на планиране .....	111
10.2    Езици за планиране на проблеми .....	112
10.2.1    Особености на езика .....	112
10.2.2    Изразителност и разширения .....	114
10.2.3    Примери.....	115
10.3    Планиране с търсене в пространството от състояния .....	117
10.3.1    Алгоритъм за планиране чрез прогресия .....	118
10.3.2    Алгоритъм за планиране чрез регресия .....	118
10.3.3    Евристики за търсене в пространството от състояния.....	119
10.4    Частично наредени планове .....	119
10.4.1    Частично наредените планове като задача за търсене.....	120
10.4.2    Примери за частично наредени планове .....	121
10.5    Речник.....	123
10.6    Ресурси.....	124
<b>11 РАЗСЪЖДЕНИЯ С НЕСИГУРНИ ЗНАНИЯ. ПРАВИЛО НА БЕЙС. УСЛОВНА НЕЗАВИСИМОСТ. ....</b>	<b>125</b>
11.1    Защо са необходими?.....	125
11.2    Как да се използва правилото в този случай?.....	126
11.3    Какви подходи са известни .....	127
11.4    Размити множества и размита логика.....	130
11.5    Видове изводи в СОЗ .....	132
11.6    Представяне на несигурни знания с вероятности .....	133
11.7    Механизми за извод .....	134
11.8    Ресурси.....	136
<b>12 БЕЙСОВИ МРЕЖИ. ....</b>	<b>137</b>
12.1    Увод в терминологията.....	137
12.2    Увод.....	138
12.3    Дефиниция .....	139
12.4    Представяне на знания в несигурна област.....	139
12.5    Видове изводи в Бейсовите мрежи .....	141
12.6    Семантика на Бейсовите мрежи .....	141
12.7    Представяне на пълното съвместно разпределение .....	141
12.8    Метод за конструиране на бейсови мрежи .....	141
12.9    Компактност и подредба на възлите .....	142
12.10    Условно независими връзки в мрежите на Бейс.....	144
12.11    Вериги на Марков .....	145
12.12    Ефективно представяне на условни разпределения .....	146
12.12.1    Бейсови мрежи с непрекъснати променливи .....	147
12.13    Приложение на Бейсовите мрежи .....	148
12.14    Речник.....	148
12.15    Ресурси.....	149
<b>13 УЧЕНЕ НА ДЪРВО НА РЕШЕНИЯТА. АНСАМБЛОВО УЧЕНЕ. ....</b>	<b>150</b>
13.1    Учене на дърво на решенията .....	150
13.2    Ансамблово учене .....	156
13.3    Речник.....	159
13.4    Ресурси.....	159

<b>14 СТАТИСТИЧЕСКИ МЕТОДИ ЗА УЧЕНЕ. НАИВЕН БЕЙСОВ МОДЕЛ .....</b>	<b>160</b>
14.1 Речник.....	160
14.2 Статистическо учене .....	161
14.3 Наивен Бейсов модел .....	163
14.4 Използвана литература .....	164
<b>15 УЧЕНЕ БЕЗ УЧИТЕЛ. КЛЪСТЕРИЗАЦИЯ.....</b>	<b>165</b>
15.1 Учене без учител .....	165
15.2 K-MEANS .....	166
15.3 Смесено Гаусово моделиране .....	170
15.4 Йерархично клъстериране.....	170
15.4.1 Йерархичен K-means.....	170
15.4.2 Agglomerative техника.....	171
15.5 Речник.....	171
15.6 Източници.....	172
<b>16 УЧЕНЕ ОСНОВАНО НА ПРИМЕРИ. РАЗСЪЖДЕНИЯ ОСНОВАНИ НА ПОДОБНИ СЛУЧАИ. ....</b>	<b>173</b>
16.1 Основни Понятия и съкращения.....	173
16.2 Какво е „учене основано на примери“? .....	174
16.3 K-NEAREST NEIGHBOR ALGORITHM .....	174
16.4 Разсъждения основани на подобни примери .....	176
16.5 Ресурси.....	181
<b>17 НЕВРОННИ МРЕЖИ.....</b>	<b>182</b>
17.1 Въведение.....	182
17.2 Биологични невронни мрежи .....	183
17.3 Основни елементи на невронните мрежи .....	184
17.4 Персептрон .....	186
17.4.1 Еднослоен персептрон.....	186
17.4.2 Многослоен персептрон.....	188
17.5 Приложения на невронните мрежи .....	191
17.6 Речник.....	191
17.7 Използвана литература .....	191
<b>18 КОМУНИКАЦИЯ. ИЗПОЛЗВАНЕ НА ФОРМАЛНИ ГРАМАТИКИ ЗА ЕСТЕСТВЕН ЕЗИК. СИНТАКТИЧЕН АНАЛИЗ.....</b>	<b>192</b>
18.1 Увод.....	193
18.2 N-ГРАМЕН МОДЕЛ.....	193
18.2.1 Определение.....	193
18.2.2 N-грамен модел върху символи .....	193
18.2.3 N-грамен модел върху думи.....	194
18.2.4 Изглеждане .....	194
18.3 Извличане на информация .....	195
18.3.1 Отговаряне на въпроси .....	195
18.3.2 Регулярни изрази .....	195
18.3.3 Автоматизирано създаване на шаблони.....	196
18.3.4 Релационно извлечане на данни.....	196
18.4 Синтактичен анализ .....	198
18.4.1 Ограничена естествен език.....	198
18.4.2 Вероятностна контекстно-свободна граматика.....	199

18.4.3	<i>Светът на Вампуса</i> .....	199
18.4.4	<i>Една граматика за светът на Вампуса</i> .....	199
18.4.5	<i>Парсване (извод, разбор)</i> .....	201
18.4.6	<i>Обучаване на вероятностите за ВКСГ</i> .....	202
18.4.7	<i>Лексикализирани ВКСГ</i> .....	202
18.4.8	<i>Съгласуване по род, число и лице</i> .....	203
18.4.9	<i>Усложнения</i> .....	205
18.5	<b>ЗАКЛЮЧЕНИЕ</b> .....	206
18.6	<b>РЕЧНИК</b> .....	206
18.7	<b>ИЗПОЛЗВАНА ЛИТЕРАТУРА</b> .....	206
<b>19</b>	<b>ПРЕДСКАЗВАНЕ ЧРЕЗ РЕГРЕСИОННИ ДЪРВЕТА</b> .....	<b>207</b>
19.1	<i>Линейна регресия</i> .....	207
19.1.1	<i>Рекурсивно разделяне</i> .....	208
19.1.2	<i>Дървата за предсказване</i> .....	208
19.2	<i>Разделяне на данните</i> .....	210
19.3	<i>Проблеми</i> .....	212
19.4	<i>Източници</i> .....	213
<b>20</b>	<b>ИЗВЛИЧАНЕ НА АСОЦИАТИВНИ ПРАВИЛА</b> .....	<b>214</b>
20.1	<i>Въведение</i> .....	214
20.2	<i>Основни понятия</i> .....	215
20.3	<i>Основни алгоритми за извличане на асоциативни правила</i> .....	215
20.4	<i>Повишаване на ефективността на алгоритмите</i> .....	216
20.4.1	<i>Намаляване на броя минавания през базата данни</i> .....	216
20.4.2	<i>Взимане на изводки</i> .....	217
20.4.3	<i>Паралелизация</i> .....	218
20.5	<i>Литература</i> .....	218
<b>21</b>	<b>МЕТОД ЗА КЛЪСТЕРИЗАЦИЯ DBSCAN</b> .....	<b>219</b>
21.1	<i>Въведение</i> .....	219
21.2	<i>Формални дефиниции</i> .....	220
21.3	<i>Коментар на дефинициите</i> .....	221
21.4	<i>Алгоритъм</i> .....	222
21.4.1	<i>Псевдокод (източник: [1])</i> .....	222
21.4.2	<i>Оценка на сложността</i> .....	223
21.4.3	<i>Параметрите Eps и MinPts</i> .....	223
21.4.4	<i>Предимства на алгоритъма</i> .....	224
21.4.5	<i>Недостатъци</i> .....	224
21.5	<i>Речник</i> .....	225
21.6	<i>Литература</i> .....	225
<b>22</b>	<b>НАМАЛЯВАНЕ НА ОБЕМА НА ДАННИ (КОМПРЕСИЯ)</b> .....	<b>226</b>
22.1	<i>Компресиране на данни</i> .....	226
22.1.1	<i>Ентропия</i> .....	226
22.1.2	<i>Ниво на компресия</i> .....	226
22.1.3	<i>Загуба на информацията</i> .....	227
22.1.4	<i>Модел на компресия</i> .....	227
22.1.5	<i>Значение за изкуствения интелект</i> .....	227
22.2	<i>Методи за компресия без загуба</i> .....	228

22.2.1	<i>Метод на Шанон-Фано</i> .....	228
22.2.2	<i>Метод на Хъфман</i> .....	228
22.2.3	<i>Метод за подобreno Хъфманово кодиране, чрез адаптивния метод</i> .....	229
22.2.4	<i>Аритметично кодиране</i> .....	230
22.2.5	<i>Кодиране чрез речници</i> .....	230
22.2.6	<i>Адаптивни речници</i> .....	231
22.2.7	<i>Други методи за кодиране</i> .....	232
22.3	<b>МЕТОДИ ЗА КОМПРЕСИЯ СЪС ЗАГУБА</b> .....	232
22.3.1	<i>Компресия на аудио</i> .....	232
22.3.2	<i>Компресия на видео</i> .....	232
22.4	<b>РЕЗЮМЕ</b> .....	233
22.5	<b>РЕСУРСИ</b> .....	233

# **1 Предмет и цели на Изкуственият Интелект.**

## **Основни направления.**

В продължение на много години сферата на Изкуственият Интелект (ИИ) се опитва да разбера как хората мислят. Това изследване се простира отвъд просто разбиране на човешката мисловна дейност, но и как да се построи система, която да мисли така. Това е сравнително нова наука започната след Втората Световна Война. Името и е дадено през 1956 година. Това е една от най-предпочитаните науки, редом с молекуларната биология. Има голямо разнообразие от подобласти - от полета с общи цели, като ученето, до специфични задачи като играене на шах, доказване на математически теореми, писане на поеми и диагностициране на заболявания.

### **Съдържание**

[Съдържание](#)

[Какво е ИИ?](#)

[Да действаш като човек: тест на Тюринг](#)

[Да мислим рационално: „закони на мисълта“](#)

[Да действаш рационално: подход на рационалния агент](#)

[Основи на изкуствения интелект](#)

[Философия](#)

[Математика](#)

[Икономика](#)

[Неврология](#)

[Психология](#)

[Компютърно инженерство](#)

[Теория на управлението и кибернетика](#)

[Лингвистика](#)

[Основни направления](#)

[Обобщение](#)

### **1.1 Какво е ИИ?**

„Изследователи в областта на изкуствения интелект се опитват да направят машини, които да проявяват поведение, което ние наричаме интелигентно поведение, наблюдавано в човешките същества.“ Slagle, James R. (1971)

Ще разгледам няколко дефиниции, които се различават по две основни измерения. Едните са свързани с мисловните процеси и мотиви, докато другите се адресират до поведение. Има

четири подхода: да действаш като човек, да мислиш като човек (формират едното измерение), да действаш рационално, да мислиш рационално (формират другото измерение). Едните измерват успеха по отношение на вярност към човешкото поведение, докато другите го измерват срещу идеалната концепция за интелигентност, която ще наричаме рационалност. Една система е рационална, ако прави "правилното нещо", като се има предвид, това, което знае. Исторически погледнато, всички четири подхода за ИИ са били следвани. Както може да се очаква, съществува известно напрежение между подходите, насочени към хората и подходите, съредоточени върху рационалността.

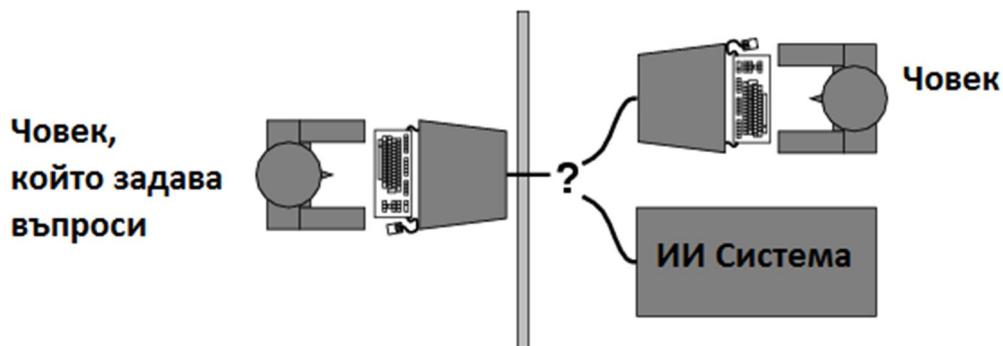
Система, която мисли като човек	Система, която мисли рационално
"Вълнуващите нови усилия, за направата на мисленци компютри ... машини с умове, в пълен и буквален смисъл." (Haugeland, 1985)	"Проучването на умствените способности чрез използване на изчислителни модели." (Chamiak и McDermott, 1985 г.)
"Автоматизация на дейностите, които ние асоциираме с човешкото мислене, дейности като: вземане на решения, решаване на проблеми, учене" (Белман, 1978)	"Изучаването на изчисления, които правят възможно да се възприема, причиня и да се действа." (Уинстън, 1992)
Система, която действа като човек	Система, която действа рационално
"Изкуството за създаване на машини, които изпълняват функции, които изискват интелигентност, когато се извършват от хора. "(Kurzweil, 1990)	"Изчислителната интелигентност е изучаването на дизайна на интелигентни агенти" (Poole et al., 1998)
"Изследването за това как да направят компютрите, таке че да правя нещца, в който хората са по-добри в момента. "(Rich и Knight, 1991)	„ИИ ... е загрижен за интелигентно поведение в артефакти. " (Nilsson, 1998)
<b>Фигура 1.1</b> Някои дефиниции на ИИ, организирани в четири категории.	

Подходът, ориентиран към човека е емпирична наука, включваща хипотеза и експериментално потвърждение. Рационалният подход включва комбинация от математика и инженерство. Всяка група отхвърля отчасти концепцията на другите, но в същото време подпомага развитието им. Нека да разгледаме четирите подхода по-подробно:

### **1.1.1 Да действаш като човек: тест на Тюринг**

Тестът на Тюринг е предложен от британския математик Алан Тюринг през 1950 г. в статията му "Изчислителни машини и разум". Тестът проверява дали компютърът има разум

в човешкия смисъл на думата. Тюринг предложил тест, който да замени безсмисления според него въпрос "Може ли машината да мисли?" с по-определен. Съвременната интерпретация на този тест изглежда по следния начин: Човек взаимодейства дистанционно с двама събеседници — компютър и човек; на основание на отговорите на въпросите, тестовият субект е длъжен да определи с кого разговаря — с машина (изкуствен интелект/компютърна програма) или с жив човек; задачата на компютърната програма е да въведе человека в заблуда, като го по този начин да направи грешен избор.



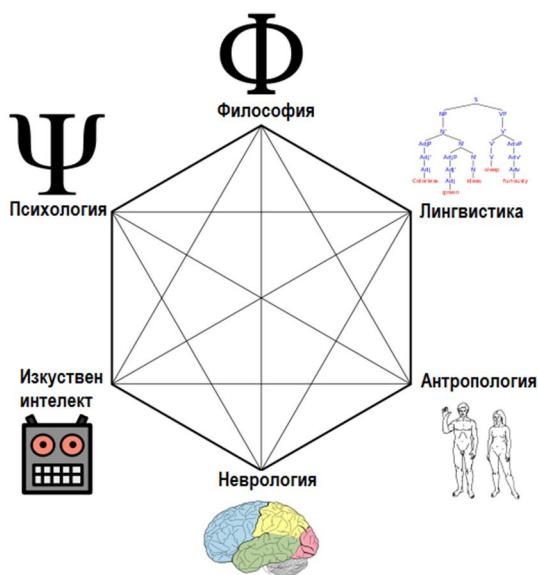
За да премине теста компютърът трябва да притежава следните възможности:

- Обработка на естествен език, за да може успешно да се общува на английски.
- Представяне на знания – за да може да съхрани това, което знае или чува.
- Автоматизирани разсъждения – използват се за съхранение на информация за отговаряне на въпроси и достигането на нови заключения.
- Машинно обучение – за адаптирането към нови обстоятелства и за засичането и откриването на нови модели.
- Визия за възприемане на обекти.
- Роботиката, за да манипулира обекти и да се движи.

Тези шест дисциплини покриват голяма част от областта на изкуствения интелект. Тестът на Тюринг е все още валиден, дори и след 50 години.

### **1.1.2 Да мислиш като човек: когнитивен (познавателен) подход за моделиране**

Ако искаме да кажем, че дадена програма мисли като човек, то ние трябва по някакъв начин да определим как мисли той. Т.е трябва да изследваме човешкия мозък. Има два начина да направим това: чрез самонаблюдение (интроспекция) – опитвайки се да хванем нашите собствени мисли по начина, по който текат – и чрез психологически експерименти. Веднъж достигнали до достатъчно прецизна теория за човешкия ум, става възможно да кодираме тази теория в компютърна програма.



Когнитивна революция е име на интелектуално течение от 1960–те, което започва като стремеж към нов подход в изследването на мозъка и процесите му и довежда до зараждането на известното сега название когнитивни науки.

Тази дисциплина се занимава с научно изследване на ума и неговите процеси. Изучава познанието - какво е и как се използва. Включва изследване на интелекта и поведението. Особен акцент се поставя върху начина, по който е представена информацията, преработена и трансформирана (като възприятие, език, памет, мислене, и емоция) в нервна система (човешки или други животински) и машини (напр. компютри). Когнитивната наука се състои от множество изследователски дисциплини, включително психология, изкуствен интелект, философия, неврология, лингвистика и антропология. Тя се простира на много равнища на анализ, от ниско ниво на обучение и механизми за вземане на решения до високо ниво логика и планиране; от невронна мрежа до модулна организация на мозъка.

### **1.1.3 Да мислим рационално: „закони на мисълта“**

Гърцкият философ Аристотел е един от първите, опитали се да класифицират „правилното мислене“ - неоспорим словесен процес. Неговият логически силогизъм осигурява модели за аргументирани структури, които позволяват достигането на правилни заключения, когато са дадени коректни начални данни. Изучаването на тези закони води до началото на нова дисциплина – Логика.

Има две основни пречки пред този подход. Първо, не е лесно да се вземат неформални знания и да се систематизират като формални условия, изисквани от логическа нотация, особено когато знанието е несигурно. Второ, има голяма разлика между това да бъдеш в състояние да решаваш проблеми "по принцип" и да правиш това на практика. Дори от проблеми само с няколко десетки факти могат да се изчерпят изчислителните ресурси на всеки компютър, освен ако няма някакви насоки, кои мисловни стъпки да опитат първо. Въпреки че тези две пречки се срещат във всеки опит за изграждане на мислещи системи, за първи път са се появили в логистичната традиция.

#### **1.1.4 Да действащ рационално: подход на рационалния агент**

Агент това е нещо, което възприема средата чрез рецептори или сензори и въздейства върху нея, чрез ефектори. В най-широк смисъл този термин обхваща хора, роботи и програми. Рационален агент е този, който действа така, че да постигне най-добър резултат. При несигурности знания, избира най-вероятният изход. Не е задължително да се включва и мисловната дейност, но ако има мисленето трябва да бъде в услуга на рационалните действия.

Рационалният агент е обект, който възприема и действа. Можем да го представим абстрактно като функция от история на възприятията до действия:  $[f: P^* \rightarrow A]$ . За всеки тип среда на действие и задачите, ние търсим агент (или група от агенти) с най-добри резултати. Ограниченията изчислителните ресурси обаче правят перфектната рационалност недостижима.

В "законите на мисълта", акцентът пада върху правилните изводи. Намиране на правилните изводи е съществена функция на рационалния агент, тъй като един от начините да действа рационално е да разсъждава логично до достигане на заключение как да постигне своите цели. От друга страна, правilen извод не винаги се постига само с рационалност, защото често има ситуации, в трябва да постъпи ирационално. Има и вградени реакции, които агентът трябва предварително да познава. Например, отдръпването от гореща печка е рефлекс за действие, който обикновено е по-успешен, отколкото по-бавно действие, предприето след внимателно изследване.

### **1.2 Основи на изкуствения интелект**

Да разгледаме накратко история на дисциплини, които са допринесли за идеи, гледни точки, и техники за ИИ. Всяка дисциплина ще представим с въпроси, на които цели да отговори, свързани с ИИ.

#### **1.2.1 Философия**

Може ли да се използват формални правила, за да се направят валидни заключения?

Как мисловната дейност възниква от физическия мозък?

Откъде идва знанието?

Как знанията довеждат до действие?

Аристотел е първият, който формулира точен набор от закони, определящи рационалната част на ума. Той развива неформална система от силогизми за правилното мислене, които позволяват да се генерираят заключения механично, като се вземат предвид, първоначалните предпоставки.

Много по-късно, Рамон Лул е имал идеята, че в действителност може да се извърши рационално мислене чрез механични артефакти.

Томас Хобс (1588-1679) предложил, че рационалното мислене е като цифровите изчисления, че "ние добавяме и изважда в напити тихи мисли."

Около 1500, Леонардо да Винчи (1452-1519), проектира, но не изгражда механичен калкулатор. Последните реконструкции са показвали, че дизайна е функционален. Първата известна сметачна машина е построена около 1623 от немския учен Вилхелм Шийкард.

Паскал пише, че "аритметичната машина произвежда ефекти, които са по-близо до мисленето отколкото всички действия на животните".

Готфрид Вилхелм Лайбниц (1646-1716) изгражда механично устройство, предназначено да извършва операции по концепции, а не по числа, но неговият обхват е по-скоро ограничен.

Сега, когато имаме идеята за набор от правила, които могат да опишат формално, рационалната част на ума, следващата стъпка е да се разгледа ума като физическа система. Рене Декарт (1596-1650) прави първата дискусия за разликата между духа и материията и проблемите, които възникват. Един от тях, с чисто физическа концепция на ума, е свободната воля. Въпреки, че Декарт е силен застъпник на силата на мотивите, той е привърженик на дуализма. Приел е, че има една част на човешкото съзнание (душа или дух), която е извън природата, освободена от физичните закони. Животните, от друга страна, не притежават това двойно качество, те биха могли да бъдат третирани като машини. Една алтернатива на дуализма е материализма, който твърди, че работата на мозъка в съответствие с законите на физиката представлява ума. Свободната воля е просто начинът, по който възприемането на наличните възможности за избор се появява в процеса на избор.

Като вече имаме ума като физическа система, която манипулира знания, следващият проблем е да се установи източник на знание. Емпиричното движение, като се започне с Франсис Бейкън (1561-1626) се характеризира с максимата на Джон Лок (1632-1704): "Нищо не е разбирано, ако не е на първо място в сетивата." Дейвид Хюм (1711-1776) е предложил това, което сега е известно като принципа на индукцията: че общите правила са придобити от излагане на повтарящи се асоциации между техните елементи.

### **1.2.2 Математика**

Какви са формалните правила за достигане на валидни заключения?

Какво може да се изчисли?

Как правим заключение с неясна информация?

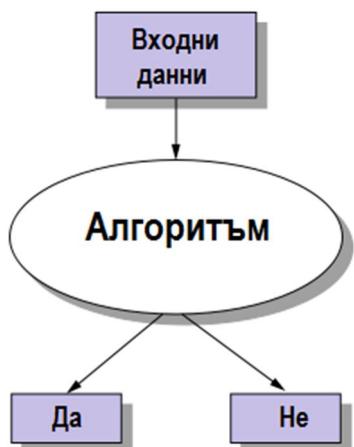
Философите са очертали най-важните идеи на ИИ, но скокът към официална наука изисква известно ниво на математическо формализиране в три основни области: логика, изчислимост и вероятности. Идеята за официална(формална) логика може да се проследи чак до древна Гърция, но математическото развитие наистина започва с Джордж Бул (1815 - 1864). Джордж Бул е британски математик и философ. Като изобретател на Буlevата алгебра, която е в основата на цялата съвременна компютърна аритметика, Бул е разглеждан като един от основателите на компютърната наука, независимо че по онова време компютри не са съществували.

Фридрих Лудвиг Готлоб Фреге (1848 - 1925) е немски математик, логик и философ. Фреге е разширил Буlevата логика като е включил обекти и релации, създател е на модерната формална логика, наричана още математическа или символна логика. Алфред Тарски (1902 - 1983) представя теория, която показва как да се съпоставят обекти в логиката на обекти от

реалния свят. Следващата стъпка е да се определи границата на достижимото чрез логика и изчислимост.

Първият нетривиален алгоритъм се смята Евклидовият алгоритъм за изчисляване на най-голям общ знаменател. Мохамед Ибн Муса ал-Хорезми дава начало на разпространението на арабските цифри в Близкия Изток и в Европа. Латинизираната форма на името Ал-Хорезми, Алгоритми, става основа на думата „алгоритъм“.

Бул и други обсъждат алгоритми за извеждане на логическо заключение, и от края на 19-ти век. През 1900, Давид Хилберт (1862 - 1943), представя списък на 23 проблема, за които той правилно прогнозира, че ще занимават математиците през по-голямата част от века. Последният проблем е свързан с въпроса дали е налице алгоритъм за извеждане на истинно твърдение от всяко логично предложение, с участнието на естествените числа – известно като „решение на проблема“. В изчислителната теория това е задача в една формална система, която връща отговор да или не.



Курт Гьодел (1960 - 1978), показва че съществува ефективна процедура, която доказва всяко вярно твърдение в логиката от първи ред (предикатно смятане от първи ред) на Фреге и Ръсел, но предикатното смятане от първи ред не може да засече принципите на математическата индукция, нужна да характеризира естествените числа. През 1931, той доказва, че съществуват сериозни ограничения. Неговата теория за непълнотата показва, че във всеки език, който е достатъчно изразителен, че да опише свойствата на естествените числа, съществуват верни твърдения, които са „нерешими“, т.е. тяхната истинност не може да бъде установена от всеки алгоритъм.

Този фундаментален резултат може да бъде интерпретиран като установяване на това, дали съществуват функции на цели числа, които не могат да се представят чрез алгоритъм – т.е. те не могат да бъдат пресметнати. Това мотивира Алан Тюринг (1912 - 1954) да опита да характеризира точно кои функции могат да бъдат изчислени.

Въпреки че неизчислимостта и нерешимостта са важни за разбирането на изчислимостта, понятието нерешимост е имало много по-голямо въздействие. Грубо казано, нерешим проблем имаме, когато времето, необходимо за решаване на част от проблема, расте експоненциално с размера й.

Освен логика и изчислимост, трета по принос в ИИ е теорията на вероятностите. Джироламо Кардано (1501 - 1576) е известен италиански ренесансов математик, лекар, философ и астролог. Той за първи път формира идеята за вероятностите, като изследва възможните резултати от хазартни събития. Вероятността бързо се превръща в безценна част от всички количествени науки. Използва се при несигурни измервания и непълни теории. Пиер дьо Ферма (1601 - 1665) и други напредват с теорията и въвеждат нови статистически методи.

Бейсовият подход, по името на Томас Бейс (1702-1761), при анализа на данни стои в основата на повечето съвременни методи на ИИ системите при наличие на неопределени знания.

### **1.2.3 Икономика**

Как трябва да се вземат решения, така че да се максимизира резултатът?

Как да действаме, когато решението не е изгодно за всички?

Как да действаме, когато резултат ще получим далеч в бъдещето?

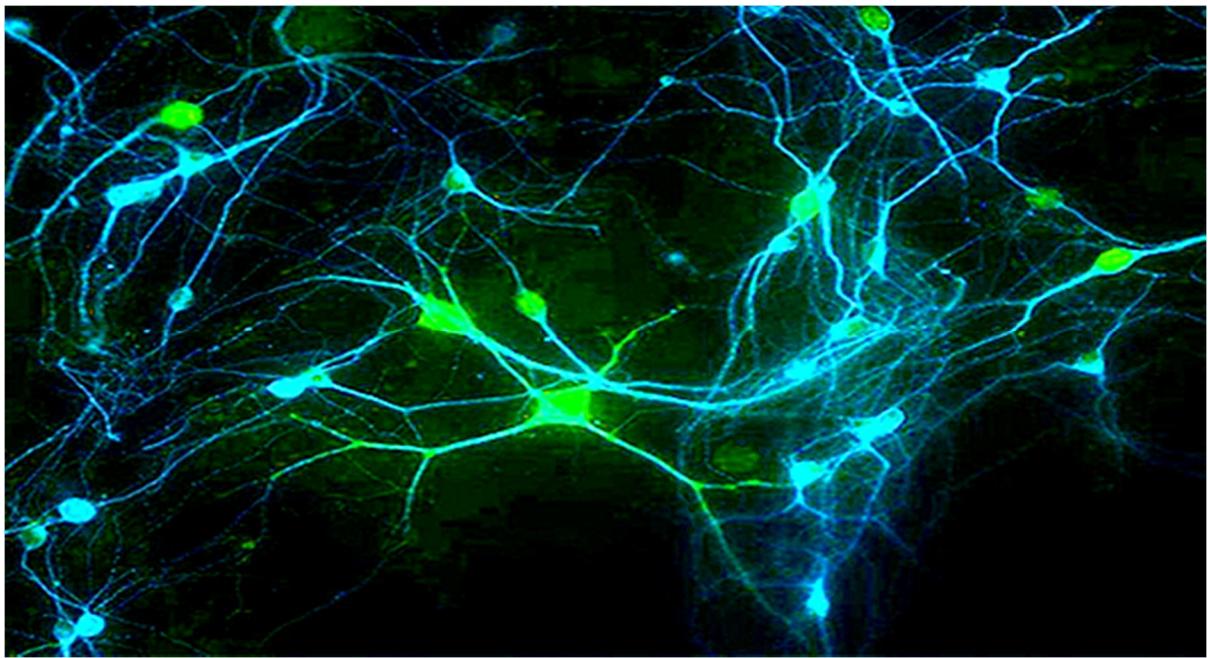
Теорията за взимане на решения съчетава в себе си теорията на вероятностите. Тя осигурява цялостна рамка за вземане на решения (икономически или други) изградена върху неясноти (несигурности).

### **1.2.4 Неврология**

Как мозъкът обработва информация?

Неврологията изучава нервната система, особено мозъка. Точният начин, по който мозъка извършила мисловната дейност е една от най-големите загадки на науката. Много отдавна е доказано, че мозъка взима участие в мисленето. Доказано е, че силен удари по главата може да доведе до умствени проблеми, т.е. възпрепятстване на мисловните дейности. Също така отдавна е известно, че човешкият мозък е по-различен от всички останали. Около 335 година, Аристотел пише: „От всички живи същества, човекът има най-голям мозък спрямо неговите размери“. Още в средата на 18 век, мозъкът е бил познат като център на съзнанието.

Пол Пиер Брока е френски лекар и патологоанатом. Той получава световна популярност с откриването на речедвигателния център в мозъка. Той се намира в третата фронтална гънка на челния дял в лявото полукълбо на мозъчната кора. Брока прави знаменитото си откритие по време на аутопсия на един пациент в известния приют за душевноболни мъже „Бисетр“ в Париж. Болестта, от която страдал пациентът е алалия. Специфичното за тази болест е, че болният не можел да говори, но въпреки това разбидал всичко и се опитвал да участва в процеса на комуникация чрез серия от нечленоразделни звуци. Пациентът остава в историята като „господин Тан“.



В един дигитален компютър изчислението е локализирано и се подчинява на много строга серия от правила. Дигиталният компютър се подчинява на законите на „машината на Тюринг“, в която има централен процесор (CPU), входяща информация и изходяща информация. Централният процесор извърши определена серия от манипулации на входящата информация и произвежда изходяща информация. Ето защо, „мисленето“ е локализирано в него.

Човешкият мозък прилича повече на обучаващца се машина, на „нервална мрежа“, която постоянно прави нови връзки, след като получи задача. Тъй като мислите са толкова дифузни и разпръснати из многото части на мозъка, може би максимумът, който учениите ще бъдат в състояние да достигнат, е да съставят речник на мислите, т.е. да установят двустренното съответствие между определени мисли и специфични модели на сканирания. Австрийският биомедицински инженер Герт Пфуртшелер например е програмирал един компютър да разпознава специфични мозъчни модели и мисли чрез съсредоточаване на усилията върху т-вълните, откривани в ЕЕГ-етата. Очевидно т-вълните са свързани с намерението за извършване на някои мускулни движения. Той казва на пациентите си да вдигнат пръст, да се усмихнат или да се намръщят и след това компютърът записва кои т-вълни са активирани. Всеки път когато пациентът извърши ментална дейност, компютърът грижливо регистрира модела на т-вълната. Този процес е труден и досаден, тъй като трябва да се отстраняват старателно лъжливите вълни, но благодарение на него Пфуртшелер открил поразителни съответствия между простите движения и определени мозъчни модели.

### **1.2.5 Психология**

Как хората и животните мислят и действат?

**Когнитивната психология** е дял от психологията, който изследва познавателните способности и процеси на човешкото съзнание и проблемите свързани с паметта, езика, логическото мислене, вниманието, възприятиета, въображението, способността за взимане на решение и др. Когнитивната психология се занимава не просто с изучаването на анатомо-

физиологичния строеж на психичната феноменология, но и изследва начините, по които структурираме и организираме придобития опит.

Уилям Джеймс (1842 - 1910) поддържа възгледа, че съзнанието се формира във връзка сприспособяването на индивида към средата. Той не вижда душата и съзнанието както те се виждат в християнството, а имат функционално значение. Според него чак след като съзнанието ни преработи дадено усещане, то става в този вид, в който ние го оствъзваме. Тъй като нашето съзнание е ограничено, ние не можем да разбираме всичко, което е около нас, а само това, което ни интересува.

### **1.2.6 Компютърно инженерство**

Как можем да изградим ефективен компютър?

За да бъде една ИИ системи успешна, си ѝ нужни две неща: интелигентност и артефакт. Компютърът е артефакта на избора. Съвременния цифров електронен компютър е изобретен независимо и почти едновременно от учени в три страни. Първият оперативен компютър е построен през 1940 г. от екипа на Алън Тюринг за една-единствена цел: дешифриране на германските съобщения. През 1943 г., същата група разработва Colossus, мощна машина с общо предназначение, базирана на вакуум.

Първият оперативен програмируем компютър е изобретение на Конрад Зусе в Германия през 1941. Зусе също е изобретил числата с плаваща запетая и е спомогнал за първата среща на високо ниво език за програмиране, Plankalkül. Първия електронен компютър, ABC, между 1940 и 1942 г. е бил сглобено от Джон Атанасов и неговият студент Клифърд Бери в Университета на щата Айова.

Оттогава до сега, все по-новият компютърен хардуер допринася за увеличаване на скоростта и капацитета и намаляване на цената. Производителността се удвоява на всеки 18 месеца.

Компютърната наука е допринесла още на ИИ със софтуерните изобретения. Например: операционни системи, езици за програмиране и инструменти, необходими за писане на съвременни програми (и документи за тях).

Вярно е и обратното. Работата в сферата на ИИ е пионер в много идеи, които са направили възможен пътят обратно към компютърните науката, включително и интерактивни преводачи, персонални компютри с прозорци и мишка и т.н.

### **1.2.7 Теория на управлението и кибернетиката**

Как могат артефакти да работят под собствения си контрол?

Ктесибий е древногръцки механик и изобретател от Александрия. Изобретява първата самоконтролираща се машина: воден часовник. **Воден часовник** е механизъм за измерване на времето чрез регулиране на потока (струята) на течност, обикновено вода. На дъното на цилиндричен или конусообразен съд се прави малък отвор, от който капка по капка изтича вода. Времето се мери в промеждущи от време, за което водата изтича. Тя трябва да изтича бавно, равномерно и напълно. Това откритие променя дефиницията за това какво може да прави един артефакт.

Норберт Винер е американски математик и логик със съществен принос в електронното инженерство и електронната комуникация, смятан за баща на кибернетиката. Той е основна фигура в създаването на **теория на управлението**. Винер е работил с Бертран Ръсел, още преди да увеличаването на интереса към биологичните и механични системи за контрол.

Модерната теория за управлението, особено клонът известен като **стохастичен оптимален контрол**, има за цел проектирането на системи, които **максимилизират целевата функция** с течение на времето. Това приблизително съвпада с нашите представи за ИИ - системи, които функционират оптимално. Тогава защо се конкурират теории от две различни области, особено като се има предвид тесните връзки между техните основатели? Отговорът се крие в непосредствената връзка между математическите техники, които са били познати на участниците, и съответните набори от проблеми, които са били включени във всеки миорглед.

### 1.2.8 Лингвистика

Как езикът се отнасят до мисълта?

Бъръс Фредерик Скинър е американски психолог, който получава докторската си степен в Харвард. През 1957 публикува книгата “Вербално поведение”. Той е считан за водещия бихевиорист и е създал една от най-основните теории за ученето на XX век. Според него ученето не може да се появи при отсъствието на някакъв вид подкреплението — положително или отрицателно.

Ноам Чомски е най-добре познат като критик на Скинър. Той публикува критиката си за “Вербално поведение” на Скинър, малко след като книгата е издадена. Неговата критика станала по-известна може би дори от самата книга. Официален отговор на критиките на Чомски не е даден от Скинър. Ноам Чомски показва как теорията за поведението (бихевиоуритъмът) не разглежда идеята на творчеството в езика - не обяснява как едно дете може да разбере и да си измисли изречения, които то никога не е чувало преди. Теорията на Чомски, която е основана на синтактични модели, връщащи ни обратно към индийския лингвист Панини.

Модерната лингвистика и ИИ се пресичат се в областта хибриди, наречена **изчислителна лингвистика** или **преработка на естествен език**. Проблемът с разбирането на естествен езика се оказа значително по-комплексен, отколкото е изглеждал през 1957 г. То изисква разбиране на предмета и контекста, а не само разбиране на структурата на изреченията. Това може да изглежда очевидно, но не е било високо ценено до 1960. Голяма част от началната работата по **представяне на знания** (изследване на това как да се сложи знания във форма, така че един компютър да ги разбира) е свързана с езика и информациите от научните изследвания по лингвистика, който са били свързани с десетилетия работа над философски анализ на езика.

## 1.3 Основни направления

Традиционните направления на изкуствения интелект са:

- Търсене в пространството от състояния

**Пространството на състоянията** се представя четворката ( $D, A, S, G$ ), където  $D$  набор на възли или състояния на графа. Те кореспондират със състоянията на процеса за търсене на решение;  $A$  множество от дъги между върховете. Те съответстват на стъпките на процеса на решаване на проблема;  $S$  непразно подмножество на  $D$ , съдържащо началното състояние на проблема.  $S \in D$ ;

- Логически извод
- Представяне и използване на знания
- Общување с КС на ограничен естествен език
- Планиране на действия
- Машинно обучение и самообучение
- Разпознаване на образи
- Компютърна алгебра
- Експертни системи
- Невронни мрежи

## 1.4 Ресурси

Stuart Russell, Peter Norvig Artificial Intelligence: A Modern Approach (2nd Edition) 2002

[http://en.wikipedia.org/wiki/B.F.\\_Skinner](http://en.wikipedia.org/wiki/B.F._Skinner)

[http://en.wikipedia.org/wiki/Verbal\\_Behavior](http://en.wikipedia.org/wiki/Verbal_Behavior)

[http://en.wikipedia.org/wiki/Paul\\_Broca](http://en.wikipedia.org/wiki/Paul_Broca)

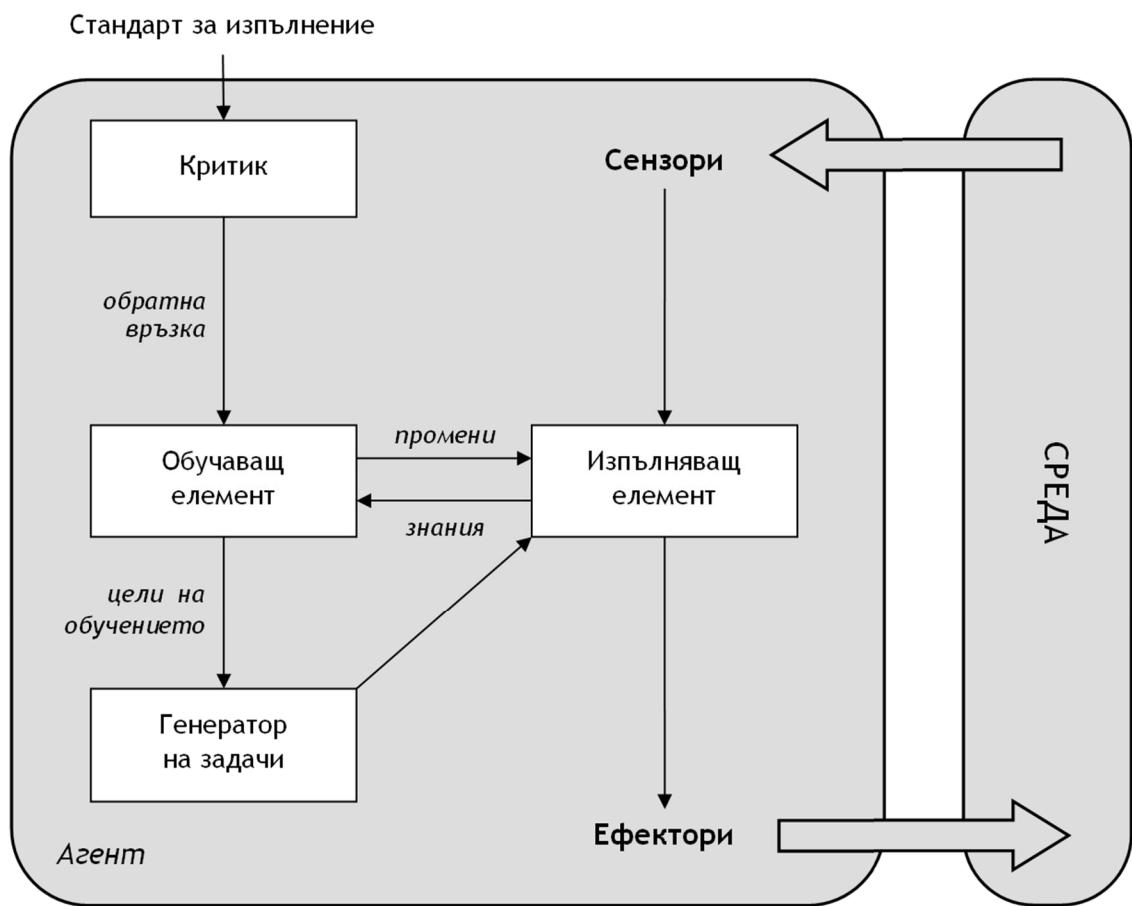
*CS461 Artificial Intelligence © Pinar Duygulu Bilkent University 2008*

<http://chitanka.info/text/17233/7>

<http://en.wikipedia.org/wiki/Ctesibius>

[http://en.wikipedia.org/wiki/Water\\_clock](http://en.wikipedia.org/wiki/Water_clock)

## 2 Интелигентни агенти



Фиг.11

## **2.1 Обобщение**

Агентът е нещо, което възприема и действа в конкретна среда. Функцията на агента изобразява всяка последователност на възприятия от средата в действия.

Оценката на ефективността измерва поведението на агента в средата. Рационалният агент се стреми да увеличи/максимизира очакваната стойност на своята оценка на ефективност, според възприятията от средата.

Средата на действие включва оценката на ефективността, околната среда, сензорите и ефекторите на агента. При проектирането на агент първата ни стъпка е да опишем максимално точно средата на действие.

Средите на действие се различават по няколко признака. Те могат да бъдат напълно или частично обозрими/достъпни, детерминистични или стохастични, епизодични или последователни, статични или динамични, дискретни или непрекъснати, едно-агентни или много-агентни.

Програмните агенти имплементират функцията на агента. Същесуват множество шаблонни програмни агенти, които отразяват как информацията, възприета или вградена, участва във

взимането на решение за следващо действие. Подходящата астратектура на програмата зависи от средата на действие на агента.

Простите рефлекторни агенти отговарят с действие на тук постъпилото възприятие, докато агентите, основани на средата, поддържат предходно състояние на средата, за да следят за възприятия, които са останали скрити за сензорите в този момент. Целенасочените агенти действат спрямо заложената им цел, а агентите, които са насочени към полезното действие, се опитват да максимилизират „щастлието“ си.

Всички агенти могат да подобрят ефективността си чрез учене.

## 2.2 Речник

<i>Agent</i>	Агент	
<i>Environment</i>	Среда	
<i>Sensors</i>	Сензорите	Рецептори
<i>Actuators</i>	Ефектори	
<i>Percept</i>	Възприятие	
<i>Percept sequence</i>	Последователност от възприятия	Верига от възприятия
<i>Agent function</i>	Функция на агент	
<i>Table of actions</i>	Таблица на състоянията	
<i>Agent program</i>	Програма на агент	Програмен агент
<i>Agent architecture</i>	Архитектура	Хардуер
<i>Rational agent</i>	Рационален агент	Разумен агент
<i>Performance measure</i>	Оценка на ефективността на поведението	
<i>Rational</i>	Рационално	Разумно
<i>Built-in knowledge</i>	Вградените знания	Предварителни знания
<i>Omniscience</i>	Всезнание	
<i>Information gathering</i>	Натрупване на знание	Събиране на информация
<i>Learning</i>	Процес на обучение	Обучение
<i>Autonomy</i>	Автономия	Самостоятелност
<i>Built-in reflexes</i>	Вградени рефлекси	Първични рефлекси
<i>Task environment</i>	Област на действие	Среда на действие
<i>Fully observable task environment</i>	Напълно достъпна среда на действие	Напълно обозрима среда на действие
<i>Partially observable task</i>	Частично достъпна среда на	Частично обозрима среда на

<i>environment</i>	действие	действие
<i>Deterministic task environment</i>	Детерминистична среда на действие	
<i>Stochastic task environment</i>	Стохастична среда на действие	
<i>Strategic task environment</i>	Стратегическа среда на действие	
<i>Episodic task environment</i>	Епизодична среда на действие	
<i>Sequential task environment</i>	Последователна среда на действие	Серийна среда на действие
<i>episode</i>	Епизоди	
<i>Static</i>	Статична среда на действие	
<i>Dynamic task environment</i>	Динамична среда на действие	
<i>Semi-dynamic task environment</i>	Полудинамична среда на действие	
<i>Discrete task environment</i>	Дискретна среда на действие	
<i>Continuous task environment</i>	Непрекъсната среда на действие	
<i>Single agent task environment</i>	Едноагента среда на действие	
<i>Multiagent task environment</i>	Многоагентна среда на действие	
<i>Competitive multi agent environment</i>	Състезателна среда на действие	
<i>Cooperative multi agent environment</i>	Кооперативна област/среда на действие	
<i>Simple reflex agents</i>	Прости рефлекторни агенти	
<i>condition-action rule, if-then rules</i>	Правило условие-действие	Правила за причинна следствена връзка
<i>Model-based reflex agents</i>	Рефлекторни агенти, основани на модели на света	
<i>Internal state</i>	Състояние на средата	
<i>Model of the world</i>	Модел на света	
<i>Goal-based agents</i>	Агенти, основани на цел	
<i>Goal</i>	Цел	

<i>Utility-based agents</i>	Агенти, основани на полезност	
<i>Utility function</i>	Функция на полезността	
<i>Utility</i>	Полезност	
<i>Learning agents</i>	Самообучаващ се елемент	
<i>Learning element</i>	Обучаващ елемент	
<i>Performance element</i>	Изпълняващ елемент	
<i>Critic</i>	Критик	
<i>Problem Generator</i>	Генератор на задачи	

## 2.3 Ресурси

Artificial Intelligence: A Modern Approach (2nd Edition) Stuart Russell, Peter Norvig 2002  
*The Many Faces of Agents.* Katia P. Sycara AAAI 1998

*Communications of the Association for Information Systems.* Ira S. Rudowsky Brooklyn College 2004  
*CS461 Artificial Intelligence* © Pinar Duygulu Bilkent University 2008

## **3 Пространство на състоянията – основни понятия и задачи**

Търсене в пространството на състоянията е процес, при който последващи състояния се разглеждат, с цел - намиране на предварително избрано(известно) крайно състояние.

### **Съдържание**

[Съдържание](#)  
[Основни понятия](#)  
[Основни дефиниции](#)  
[Представяне на пространството на състоянията](#)  
[Действия, свързани с пространството на състоянията](#)  
[Генериране на състояния](#)  
[Оценка на състояние](#)  
[Дефиниране на пялото пространство на състоянията](#)  
[Основни типове задачи, свързани с пространството на състоянията](#)  
[Основни задачи](#)  
[Агент, основан на решаването на задача](#)  
[Формулиране на целта](#)  
[Формулиране на проблема](#)  
[Търсене на решение](#)  
[Добре дефинирани задачи и решения](#)  
[Начално състояние](#)  
[Възможни действия](#)  
[Оценка за целево състояние](#)  
[Цена на пътя](#)  
[Формулиране на задача](#)  
[Примерни задачи](#)  
[Учебни задачи](#)  
[Проблеми от реалния свят](#)  
[Търсене на решения](#)  
[Резюме](#)  
[Пространство на състоянията. Основни понятия и задачи | Речник](#)  
[Пространство на състоянията. Основни понятия и задачи | Ресурси](#)

### **3.1 Основни понятия**

Решаването на интелектуални задачи, може да бъде сведено до преминаване през различни състояния, като всяко следващо е или по-просто или еквивалентно на предходното, докато се достигне до целево състояние, което се счита за решение на задачата. Такива задачи са преобразуванията на тригонометрични и алгебрични задачи, решаване на уравнения, задачи от областта на интелектуалните игри и други.

#### **3.1.1 Основни дефиниции**

*Състояние* е формулировка на задачата в процеса на нейното решаване.

*Видовете състояния* могат да бъдат начално, междинни и крайни(които можем да наричаме също целеви).

*Операторът* е правило, алгоритъм, функция, връзка, или друг начин(способ), по който едно състояние се получава от друго.

*Пространство на състоянията* е съвкупността от всички възможни състояния, които могат да се получат от дадено начално състояние.

### **3.1.2 Представяне на пространството на състоянията**

Възможни са различни представления на пространството на състоянията като представяне чрез низ, списък, на естествен език. Въпреки това, най-естественият начин за представяне на пространството на състоянията е, чрез ориентиран граф с възли представящи състоянията, и ориентирани дъги представящи операторите. Възможно е пространството на състоянията да бъде и дърво, в такъв случай имаме дърво на състоянията.

## **3.2 Действия, свързани с пространството на състоянията**

### **3.2.1 Генериране на състояния**

Генерирането на състояния обикновено се осъществява или с генериране на следващ наследник, или с генериране на всички наследници на текущото състояние. В процеса на генериране се стига до два типа състояния. Едните са такива, от които могат да се генерират още състояния-наследници, другите са такива, от които или няма смисъл да се генерират наследници, или няма наследници, или са генериирани всички наследници, или сме достигнали целево състояние.

### **3.2.2 Оценка на състояние**

Дадено състояние се оценява, спрямо критерии. В общия случай се оценява, дали текущото състояние е целево, или не е. Възможно е да има въведена метрика, което означава, че ще се установи степента на близост на текущото състояние до желаната цел(целевото състояние).

### **3.2.3 Дефиниране на цялото пространство на състоянията**

В много случаи се генерира само част от пространството на състоянията, тъй като е възможно достигането до целта да се случи много преди да се генерира цялото пространство на състоянията, което обезмисля генерирането на останалата част от пространството. Възможно е пространството на състоянията да е прекалено голямо и цялостното му генериране би довело до различни по естество проблеми.

### **3.2.4 Основни типове задачи, свързани с пространството на състоянията**

В тази графа влизат генериране на пространство на състоянията, решаване на задачи върху генерирано пространство на състоянията(търсене на цел, търсене на минимален път до цел, намиране на печеливша стратегия), комбинирана задача(едновременно постепенно

генериране на пространство на състоянията и оценка на генерираните до даден момент състояния).

### **3.3 Основни задачи**

#### **3.3.1 Агент, основан на решаването на задача**

Ще се запознаем как агент може да намери последователност от действия, които постигат целево състояние, въпреки че нико едно от действията само не може да постигне целта. Тези агенти са базирани на агентите, основани на цел. Агентите, основани на решаването на задача, избират какво да направят като намират последователности от действия, които водят до желани състояния. Ще започнем като дефинираме елементите, конструиращи даден проблем и неговото решение, и ще дадем няколко примера за пояснение.

Производителността на агентите понякога може да бъде увеличена, ако агентът успее да се адаптира към цел и да се опита да я постигне. Нека си представим един агент в Арад, Румъния. Нека си представим, че агентът трябва да стигне до Букурещ за определено време. В такъв случай агентът трябва да адаптира целта да отиде в Букурещ. Действия, които не позволяват на агента да стигне навреме могат да бъдат премахнати, по този начин проблемът със вземането на решение от агента е опростено.

#### **3.3.2 Формулиране на целта**

Целите подпомагат организирането на поведението като се намаляват целите, които агентът се опитва да постигне. Формулирането на целта, предвид текущата ситуация и производителността на агента, е първата стъпка в решаването на задачата. Задачата на агента е да разбере кои последователности от действия ще доведат до целево състояние. Преди да може да реши обаче, трябва да избере какви действия и състояния да приема. Прекалената детайлност добавя голяма доза несигурност, тъй като има прекалено много стъпки в решението.

#### **3.3.3 Формулиране на проблема**

Формулирането на проблем е процес на предприемане на действия и приемане на състояния, спрямо цел. Засега ще приемем, че нашият агент смята за действие(операция) пътуването от един град в друг. Състоянието разбира се са самите градове.

Агентът имам адаптирана задача да стигне до Букурещ и трябва да реши накъде да тръгне от началното състояние, а именно Арад. Има три пътя от този град към Сибиу, към Тимишоара и към Зеринд. Нито един от тези градове не постига целта и най-вероятно агентът не знае накъде да тръгне. Най-доброто което може да направи е да избере една от дестинациите на прозиволен принцип. Да предположим, че агентът има карта. С нея той разполага с информация за състоянието, към които може да се насочи и действията които може да предприеме. Агентът може да използва информацията за трите състояния(трите града), за да избере състояние, от което да достигне целево състояние. След като е намерил път от началното(Арад) до крайното(Букурещ) състояние, единствено му остава да премине по вече намерения път. Казано накратко, агент с няколко възможни неизвестни състояния може да

изследва последователности от възможни действия, които водят до известни състояния и след това да избере най-добрата последователност от действия.

### 3.3.4 Търсене на решение

Процесът на изследване за най-добрата последователност от действия се нарича търсене. Алгоритъм за търсене приема дефинирана задача и връща решение под формата на последователност от действия. След като е намерено решение, препоръчаните действия могат да бъдат изпълнени. Това е фазата на изпълнение на решение. След формулирането на цел и задача за решаване, агентът извиква процедура за търсене, за да реши задачата. След това използва намереното решение, за да напътства действията си. Щом решението е изпълнено, агентът ще започне формулирането на нова цел.

На фигура 1. е показан агент, основан на решаването на задача. Първо формулира задача и крайна цел, търси последователност от действия, които ще решат задачата и след това изпълнява действията едно по едно. Когато приключи формулира друга цел и започва отново. Редно е да отбележим, че при изпълнение пренебрегва своите възприятия, приема че решението, което е намерили ще работи винаги.

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  inputs: percept, a percept
  static: seq, an action sequence, initially empty
          state, some description of the current world state
          goal, a goal, initially null
          problem, a problem formulation

  state  $\leftarrow$  UPDATE-STATE(state, percept)
  if seq is empty then do
    goal  $\leftarrow$  FORMULATE-GOAL(state)
    problem  $\leftarrow$  FORMULATE-PROBLEM(state, goal)
    seq  $\leftarrow$  SEARCH(problem)
    action  $\leftarrow$  FIRST(seq)
    seq  $\leftarrow$  REST(seq)
  return action
```

Фигура 1.

Ще опишем първо процеса на формулиране на задачата. Агентът от фигура 1. по дизайн приема, че средата е статична, тий като формулирането и решаването на задачата се прави без да се обръща внимание на промени, които могат да настъпят в средата. Агентът също предполага, че началното състояние е познато. Изследването на алтернативни възможни действия, предполага средата да бъде разглеждана като дискретна. Най-важното е, че агентът приема средата за детерминирана. Решенията на задачите са прости последователности от действия, така че те не могат да се справят с неочеквани събития, още повече решенията се изпълняват без да се вземат предвид. Всички тези предположения означават, че агентите се занимават с най-лесните видове среди.

## **3.4 Добре дефинирани задачи и решения**

Една задача се състои от четири компонента: начално състояние, действия(операции), оценка, дали състоянието е целево, и цена.

### **3.4.1 Начално състояние**

Това е състоянието от което агентът започва работата си. Например началното състояние на агента в Румъния може да се опише като B(Арад).

### **3.4.2 Възможни действия**

Описание на възможните действия, които агентът може да изпълни. Най-често се използва функция на наследник. В конкретно състояние, функцията връща множество наредени двойки (действие, наследник), където всяко действие е действие на текущото състояние и всеки наследник е състояние, което може да бъде достигнато от текущото прилагайки действието. Например от състояние B(Арад), функцията може да върне една от трите двойки състояния (Към(Сибиу), B(Субиу)), (Към(Тимишоара), B(Тимишоара)), (Към(Зеринд), B(Зеринд)). Началното състояние и функцията наследник дефинират пространство на състоянията - множеството от всички състояния достигими от началното. Пространството на състоянията образуват графа, в който градовете са възлите, а дъгите между възлите са действията. (Фигура 2. може да интерпретираме като граф на пространството на състоянията) Път в пространството на състоянията е последователност от състояния свързани от последователност от действия.

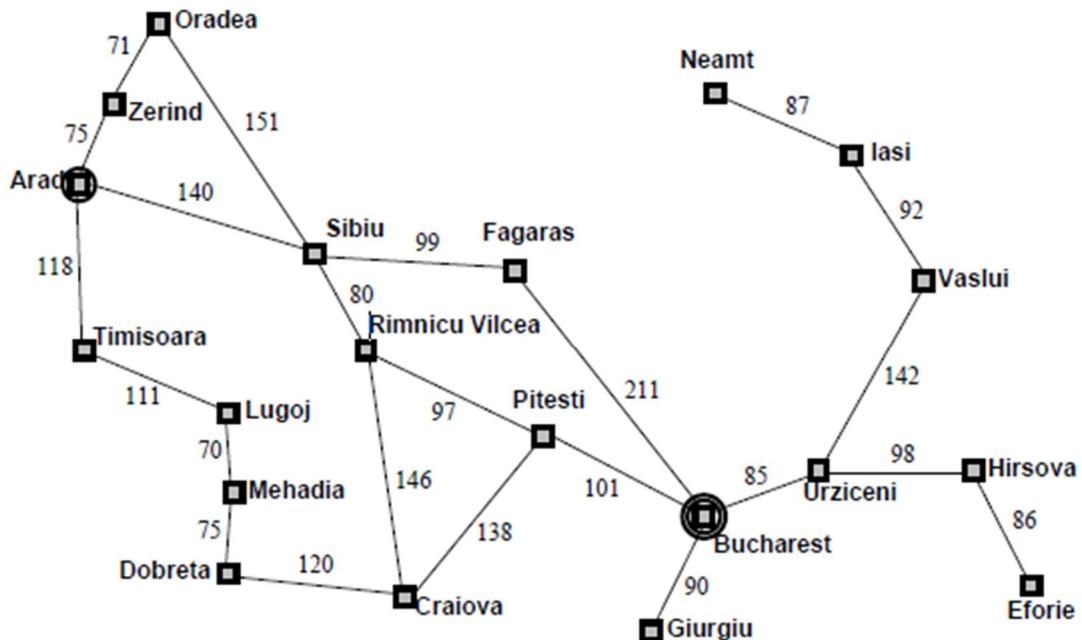
### **3.4.3 Оценка за целево състояние**

С тази оценка решаваме, дали дадено състояние е крайно (целево). Възможно е целевите състояния да са изрично зададени, тогава оценката е просто, дали състоянието е едно от целевите. Целта на агента в Румъния е B (Букурешт). Понякога обаче целевото състояние е зададено с абстрактни свойства. Например в шаха, целта е да достигнем състояние “шах и мат”.

### **3.4.4 Цена на пътя**

Оценяването на пътя се извършва с функция, която задава цифрова стойност на всеки път. За агентът, който е опитва да стигне до Букурешт, е важно да стигне колкото се може по-бързо, затова цената на пътя може да е дължината в километри. Ще представим цената на пътя като сума на отделните действия по пътя. На фигура 2. са показаните цените на стъпките като разстояния.

Предходните елементи дефинират задача и могат да бъдат събрани в една структура от данни, която да се подаде като входен параметър на алгоритъм за решаването на задачата. Решението на задачата е път от началното състояние до целево такова. Качеството на решението се определя от функцията за цената на пътя, като оптимално решение е това с най-ниска цена измежду всички възможни решения.



Фигура 2. Опростена пътна карта на част от Румъния.

### **3.4.5 Формулиране на задача**

В предната секция формулирахме задача за достигане до Букурещ като дефинирахме начално състояние, функция-наследник, оценка за цел и цена за път. Формулировката изглежда смислена, но пропуска много аспекти от реалния свят. Описанието за състоянието В(Арад) сравнено с реалния свят, където има компания с която пътувате, кое радио ще пуснете, външния пейзаж, дали има служители на реда, колко далече е следащото място за почивка, пътната обстановка, времето и други, изглежда доста опростено. Всички изброени съображения не са взети предвид, тъй като те са незначителни за намирането на решение на задачата. Този процес, на премахване на детайли, наричаме *абстракция*.

В допълнение освен абстракция в описанието на състоянието, трябва да добавим абстракция и в действията. Действието каране има много ефекти. То промения местоположението си, отнема време, изразходва гориво и други. В нашата формулировка вземаме предвид само промяната на местоположението. Също така ще пропускаме и други действия като включване на радиото, гледане през прозореца, намаляване покрай служители на реда и други.

Дали можем да бъдем по-прецизни при избирането на нивото на абстракция? Да помислим за абстрактните състояния и действия, които сме избрали, като кореспондиращи на голямо множество от състояния от детайлния свят и последователности от подробни действия. Сега да разгледаме решение на абстрактен проблем: например, път от Арад през Сибиу, Римнику Вилча, Питести до Букурещ. Това абстрактно решение съответства на голям брой детайлзиирани пътища. Например, ние можем да караме с включено радио между Сибиу и Римнику Вилча и след това да изключим радиото за останалото пътуване. Абстракцията е валидна, ако можем да разширим всяко абстрактно решение в свят с повече детайли; достатъчно условие е това за всяко подробно състояние  $B(\text{Арад})$  да има подробен път до състояние, което е  $B(\text{Сибиу})$  и така нататък. Абстракцията е полезна, ако извършването на

всяко от действията в решението е по-лесно от оригиналната задача; в този случай те са достатъчно лесни да можем да ги изпълним без по-нататъчно търсене или планиране. Изборът на добра абстракция включва премахването колкото се може повече детайли, докато запазваме валидността и абстрактните действия да са лесни за изпълнение.

## 3.5 Примерни задачи

Подходът за решаването на проблеми се прилага за голям брой среди. Ще разгледаме някои по-известни и ще ги разделим на учебни задачи и на проблеми от реалния свят. Учебната задача е предназначена да покаже или да се упражни различни методи за решаване на проблеми. На нея може да се даде точно описание. Това значи, че лесно може да се използва от различни разработчици, за да се сравни изпълнението на алгоритмите. Проблем от реалния свят е такъв, че хората се интересуват от неговото решение. Обикновено нямат ясно определено описание.

### 3.5.1 Учебни задачи

#### Осемте плъзгащи се плоочки

Осемте плъзгащи се плоочки, показано на фигура 3., се състои от 3x3 дъска с осем номерирани плоочки и празно поле. Плочките в непосредствена близост до празното поле могат да се плъзгат на празното поле. Целта е да достигнем целево състояние, което е показано отляво на фигура 3. Стандартната формулировка е следната:

- състояния - описането на състоянията съдържа мястото на всяка една от плочките и празното поле в деветте полета.
- начално състояние - всяко състояние може да бъде определено за начално. Имайте предвид, че дадена цел може да бъде постигната от точно половината от възможните първоначални състояния.
- функция-наследник - генерира редовни състояния, които се получават при преместване на празното поле наляво, надясно, нагоре и надолу.
- проверка за целево състояние - проверява дали състоянието съвпада с целевата конфигурация показана на фигура 3.(възможни са и други целеви конфигурации).
- цена на пътя - всяка стъпка струва 1, така че цената са стъпките направени в пътя.

Да разгледаме какви абстракции сме приели тук. Действията са абстрагирани до самото начало а финалните състояния, пренебрегваме междуинните места, когато празното поле се мести. Абстрагирали сме се от действия като разтрисане на дъската, когато плочките се заклещят, или изваждането им с някакъв инструмент и слагането им обратно. Останали сме с описание на правилата на пъзела, избягвайки всички детайли при физическо манипулиране.

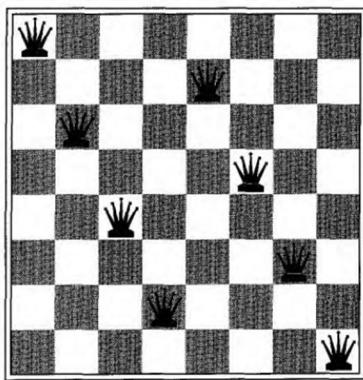


Фигура 3.

Осемте плъзгащи се плочки принадлежи на семейството от пъзели с плъзгащите се плочки, които често се използват като тестови задачи за нови алгоритми за търсене в Изкуствения Интелект. Този клас е известен, че е NP-пълен, така че не се очаква да се намерят методи значително по-добри в най-лошия случай от алгоритмите описани тук. Осемте плъзгащи се плочки има  $9! / 2 = 181\,440$  достигими състояния и е лесно решимо. Петнадесетте плъзгащи се плочки(4x4 дъска) има близо 1.3 трилиона състояния и произволни инстанции могат да бъдат решени оптимално за няколко милисекунди от най-добрите алгоритми. Двадесеттеичетири плъзгащи се плочки(5x5 дъска) има към 10 на 25-та степен състояния и произволни инстанции са трудни за решаване оптимално с днешните машини и алгоритми.

### Задача за осемте дами

Целта на осемте дами е да се поставят осем дами на шахматната дъска, така че никоя да не може да атакува друга(Дамата може да атакува фигура в същата колона, ред или диагонал). Фигура 4. показва решение, което не е правилно: дамата в най-дясната колона е атакувана от тази в най-горната лява колона.



Фигура 4.

Въпреки че съществуват специални алгоритми за тази задача и за цялата фамилия за n-те дами, остава интересна задача за алгоритмите за търсене. Има две главни формулировки. Инкременталната формулировка включва оператори, които подобряват описанието на състоянието, като се започва с празно състояние; за осемте дами това означава, че всяко действие добавя дама към състоянието. Другата формулировка започва с осемте дами на дъската и ги размества. И в двата случая цената на пътя не е от интерес, защото само

финалното състояние се брои. Ето една инкрементална формулировка която би могла да се пробва:

- състояния - всяко разположение на 0 или не повече от 8 дами на дъската
- начално състояние - няма дами на дъската
- функция-наследник - добавяне на дама на празно квадратче
- цел - осемте дами да са на дъската, никој една да е атакувана

В тази формулировка имаме  $64 \times 63 \times \dots \times 57$  възможни последователности за изследване. Погодбата формулировка би забранила да добавяме дама на място, което може да бъде атакувано:

- състояния - разположение на 0 или не повече от 8 дами, една в колона в най-лявата възможна, и никој една дама да не е атакувана
- функция-наследник - добавяне на дама на някое квадратче в най-лявата възможна колона, така че да не е атакувана от друга дама

Тази формулировка редуцира пространството на състоянията до 2057 и решението са лесни за намиране.

### **3.5.2 Проблеми от реалния свят**

#### **Търсене на маршрут**

Алгоритмите за намиране на маршрут се използват в различни приложения, като маршрутизация в компютърни мрежи, планиране на военни операции, системи за навигация на въздушното пространство. Тези проблеми са сложни за специфициране. Да разгледаме опростен пример за маршрут на въздушните линии:

- състояния - всяко е представено чрез място(летище) и текущото време
- начално състояние - специфицира се от задачата
- функция-наследник - връща състоянията получени от всеки планиран полет, които излита след текущия час плюс времето за полет от текущото летище до другото
- проверка за целево състояние - дали сме на дестинацията за никакво предефинирано време?
- цена на пътя - зависи от цената на полета, времето за чакане, митница, време на денонището и други

Системите за поддръжка на комерсиални полети използват подобна формулировка със възможност да се справят с много допълнителни усложнения.

#### **Задача за търговския пътник**

Задачата за търговския пътник е проблем, в който вски град трябва да бъде посетен точно веднъж. Целта е да се намери най-краткия път. Голямо количество от труд е положено, за да се подобрят възможностите на алгоритмите решаващи тази задача. В допълнение към

планирането на пътя, тези алгоритми се използват за задачи като планиране движението на складиращите машини по етажите.

### **Навигация на робот**

Навигацията на робот е обобщена задача за намиране на път. Вместо дискретно множество от пътища, робот може да се движи с безброй много възможни действия и състояния. За робот движещ се по плоска повърхност, пространството е двумерно. Когато роботът разполага с ръце и крака, възможното пространство за търсене става многомерно. Много напреднали техники са нужни, за да се направи пространството за търсене крайно. Като добавка към сложността на задачата, истинските роботи трябва също да се справят с грешки в сензорите и в двигателните си контролери.

Последователностите от действия при автоматичния монтаж на сложни обекти от робот бе демонстрирано от ФРЕДИ през 1972. Прогресът по-нататък е бавен, но едно е сигурно, монтажът на сложни части като електрически мотори е икономически изгодно. При монтажа ръката на робота трябва да намери реда, при който да сглоби частите на някакъв детайл. Ако се избере грешен ред, няма да може да се добавят други части, без да се премахнат някои сложени преди това. Проверка за стъпка в последователността от действия е трудна геометрична задача за търсене, тясно свързана с навигацията на роботите. Друга задача за автоматично сглобяване е дизайн на протеини, където целта е намирането на последователност от аминокиселини, които да се слеят в триизмерен протеин, притежаващ определени свойства срещу болести.

В последните години се увеличи нуждата от софтуерни роботи, осъществяващи търсене в интернет мрежата, където да търсят отговори на определени въпроси, информация, или за пазарни сделки. Това е удобна среда за алгоритми за търсене, защото е лесно да се представи интернет пространството като граф с възли страници свързани с хипервръзки

## **3.6 Търсене на решения**

След като формулирахме няколко задачи, сега ни трябва начин, по който да ги решим. Това става като търсим в пространството на състоянията. В общия случай може да имаме граф, вместо дърво, и може да минаваме през дадено състояние повече от веднъж.

Фигура 5. показва развития на дървото за търсене на маршрут от Арад към Букурешт. Коренът на дървото е възел, съответстващ на началното състояние, B(Арад). Първата стъпка е да се провери, дали това е целево състояние. Очевидно то не е, но е важно да разрешим тънки проблеми като “тръгваме от Арад, отиваме в Арад”. Тъй като това не е целево състояние, трябва да разгледаме други състояния. Това се получава като развием текущото състояние; прилагаме функцията-наследник върху текущото състояние, по този начин генерираме ново множество от състояния. В този случай, получаваме три нови състояния: B(Сибиу), B(Тимишоара), B(Зеринд). Сега трябва да изберем някоя от тези възможности, за да продължим.

Това е същността на търсения - следваме една възможност и оставяме останалите засега, в случай че първият избор не доведе до решение. Нека изберем Сибиу първо. Проверяваме, дали това е целево състояние(то не е), след това го развиваме и получаваме B(Арад),

В(Фагарас), В(Орадеа) и В(Римнику Вилча). Сега можем да изберем всяка от тези четири, или да се върнем назад и да изберем Тимишоара или Зеринда. Продължаваме, избирайки, проверявайки и развиваийки, докато или намерим решение, или няма повече състояния за развитие. Изборът на състояние се определя от стратегията за търсене.

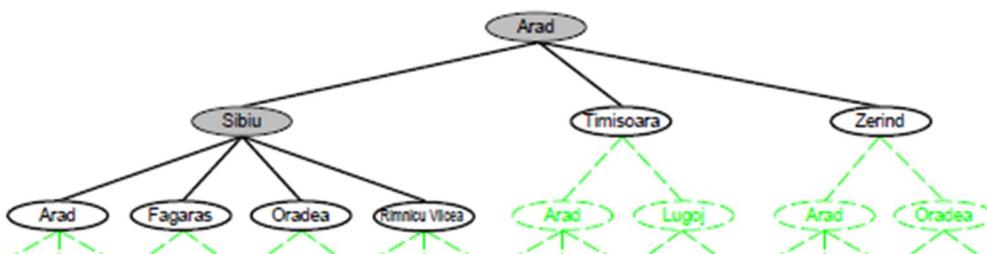
Важно е да правим разлика между пространство на състоянията и дървото за търсене. В задачата за намиране на път, има само 20 състояния в пространството, едно за всеки град. В същото време има безброй много пътища в пространството на състоянията, затова дървото за търсене има безкраен брой възли. Например трите пътища Арад-Сибиу, Арад-Сибиу-Арад, Арад-Сибиу-Арад-Сибиу са само три от безкрайна последователност от пътища (Разбира се добритите алгоритми пропускат подобни пътища).

Възможни са много начини за описание на възлите, но нека приемем, че възел съдържа следните пет компонента:

- състояние
- възел-родител
- действие
- цена
- дълбочина - брой стъпки от началното състояние

Важно е да различаваме възел от състояние.

Нужно ни е също да представим колекцията от възли, които са генериирани, но не са все още развити - тази колекция ще наричаме фронт. Всеки елемент на фронта е листо.



Фигура 5. Частично дърво за търсене на път от Арад към Букурещ. Развитите възли са изобразени със сив фон; възли, които са генериирани, но не са развити, са с бел фон; възли, които все още не са генериирани са заградени със зелен пунктир.

На фигура 5. фронтът се състои от възлите заградени с удебелен овал. Стратегията за търсене трябва да избира следващият възел, който ще бъде развит. Изчислението може да се окаже доста скъпо, тъй като функцията трябва да обходи множеството от състояния, за да избере най-доброто.

### 3.7 Резюме

Търсене в пространството на състоянията е процес, при който дадени състояния се разглеждат, докато се достигне до целево състояние, което се счита за решение на задачата. Със състоянията са свързани основните действия генериране на състояния и оценка на състояние. Върху пространството на състоянията чрез определени методи агент може да решава какви действия да предприеме върху позната и статична среда. Агентът може да конструира последователности от действия, с които да постигне целите си. Този процес се нарича търсене. Преди да започне да търси решение, агентът трябва да формулира цел и след това да формулира задача. Задачата се състои от начално състояние, оценка за цел и цена. Средата на задачата представлява пространството на състоянията. Пътят от началното състояние до целевото в пространството на състоянията е решение. Алгоритъм за търсене в дърво може да реши всяка задача. Различните варианти на алгоритъма предполагат различни стратегии.

### 3.8 Речник

action	действие/операция
action sequence	последователност от действия/операции
depth	дълбочина
expand	развитие
failure	неуспех
goal	цел
goal state	целево състояние
initial state	начално състояние
percept	възприятие
problem	проблем/задача
solution	решение
start state	начално състояние
state	състояние
strategy	стратегия

### 3.9 Ресурси

Artificial Intelligence: A Modern Approach (2nd Edition) Stuart Russell, Peter Norvig 2002

CS461 Artificial Intelligence © Pinar Duygulu  
Bilkent University 2008



## **4 Методи на 'сляпо' търсене на път до определена цел**

Чрез които виждаме как един агент може да намери поредица от действия, които постигат желания резултат, когато само едно действие не е достатъчно.

### **Съдържание**

[Съдържание](#)

[Обща постановка](#)

[Търсене в широчина \(Breadth-first search\)](#)

[Алгоритъм](#)

[Псевдокод](#)

[Свойства](#)

[Търсене с непроменяща се цена \(Uniform-cost search\)](#)

[Алгоритъм](#)

[Псевдокод](#)

[Свойства](#)

[Търсене в дълбочина \(Depth-first search\)](#)

[Алгоритъм](#)

[Псевдокод](#)

[Свойства](#)

[Ограничено търсене в дълбочина \(Depth-limited search\)](#)

[Алгоритъм](#)

[Псевдокод](#)

[Свойства](#)

[Итеративно търсене в дълбочина \(Iterative deepening depth-first search\)](#)

[Алгоритъм](#)

[Псевдокод](#)

[Свойства](#)

[Двупосочко търсене \(Bidirectional search\)](#)

[Алгоритъм](#)

[Свойства](#)

[Заключение](#)

[Методи на 'сляпо' търсене на път до определена цел | Ресурси](#)

## 4.1 Обща постановка

*Решаването на много задачи, традиционно смятани за чисто интелектуални, може да бъде сведено до последователно преминаване от едно описание на задачата към друго, еквивалентно на тървото или по-просто от него, докато се стигне до това, което се смята за решение на задачата.*

Този раздел обхваща шест стратегии за търсене, които идват под заглавието неинформирано търсене (наричан още „сляпо“ търсене). Общата задача е да се намери дадено състояние, като представяме всички възможни състояния и преходите между тях като дърво. Терминът “сляпо” означава, че стратегията няма допълнителна информация за състояния, извън предвидените в дефинирането на проблема. Всичко, което тя може да направи, е да генерира наследници и да прави разлика между целевото състояние и междинното състояние. Всички стратегии за търсене се различават от реда, в който се разширява пътят им с добавянето на възли.

Стратегиите, които ще разгледаме са:

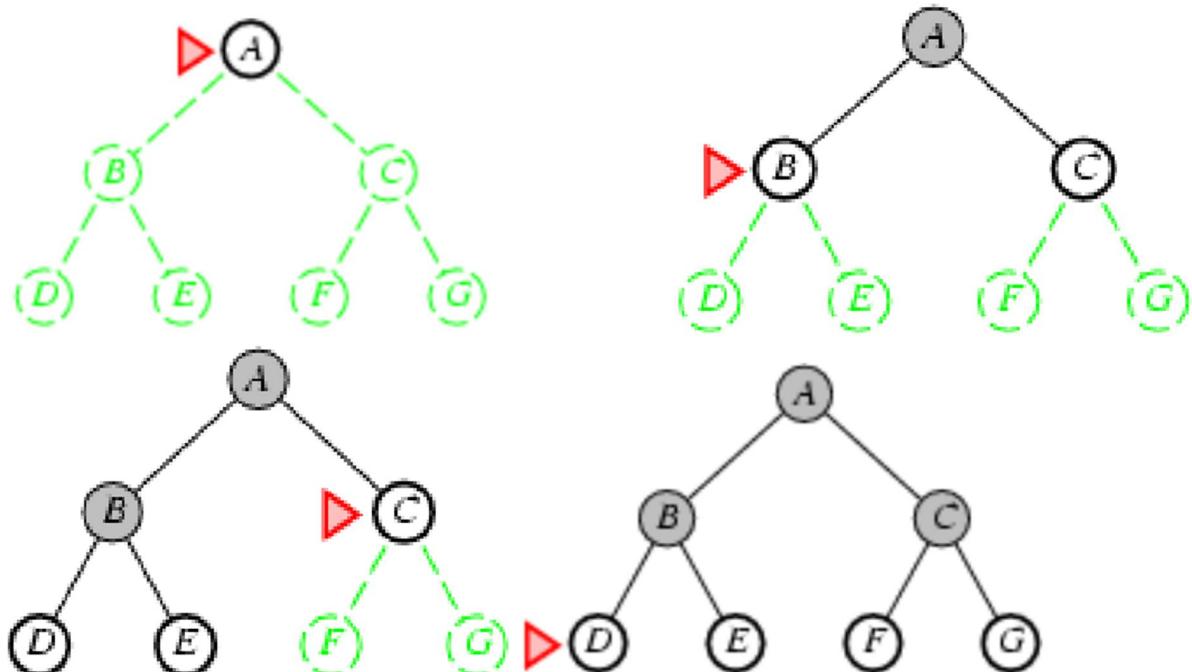
- -Търсене в широчина (Breadth-first search)
- -Търсене с непроменяща се цена (Uniform-cost search)
- -Търсене в дълбочина (Depth-first search)
- -Ограничено търсене в дълбочина (Depth-limited search)
- -Итеративно търсене в дълбочина (Iterative deepening search)
- -Двупосочно търсене (Bidirectional search)

## 4.2 Търсене в широчина (Breadth-first search)

Търсенето в широчина е проста стратегия, при която първо се разглежда коренът, след това всички наследници на корена, както и техните наследници и т.н. Или казано с други думи, първо се обхождат всички възли на дадено ниво, преди да се разгледат възлите на по-долно ниво.

Стратегията може да се имплементира като се извърши търсене в дърво върху празна структура, която трябва да е от типа First-in-first-out (примерно опашка). Това гарантира, че възлите, които са били първо посетени, ще бъдат разгледани първи, тъй като новогенерираните наследници, ще се намират на края на опашката. По този начин по-плитките възли ще бъдат разгледани преди по-дълбоките.

## Алгоритъм



Фигура 1.

От Фигура 1. лесно можем да видим, че стратегията е пълна – ако търсеният най-плитък възел е на някаква крайна дълбочина, алгоритъмът ще го намери след като премине през всички по-плитки възли (вземайки предвид, че разклонеността на дървото  $b$  е крайна). Обаче не е задължително най-плиткият възел да е оптимален, тъй като погледнато от техническа гледна точка това е така, само когато цената на пътя е функция, която не е намаляваща спрямо дълбочината на възела.

Въпреки качествата си, тази стратегия не е винаги за предпочитане, тъй като трябва да вземем предвид количеството памет и време, необходими за пълно търсене. За да получим по-точна представа за необходимите ресурси ще разгледаме хипотетична ситуация, при която за всяко състояние имаме  $b$  възможности ( $b$  на брой наследници на всеки възел или разклоненост на дървото). Коренът на дървото генерира  $b$  възела, всички на първо ниво. Всеки от тях генерира съответно по нови  $b$  възела, или общо  $b^2$  възела на второ ниво и т.н. Сега да допуснем, че търсеният възел на ниво  $d$ . Така ще разширим всички, освен последният възел на ниво  $d$ , генерирайки  $b^{d+1}-b$  възела на ниво  $d+1$ . Така общият брой на възлите става:

$$b+b^2+b^3+\dots+b^d+b^{d+1}-b = O(b^{d+1})$$

Всеки генериран възел трябва да остане в паметта, т.к. или е част от пътя или наследник на възел от пътя. Поради тази причина пространствената сложност е същата, като времевата сложност (плюс един възел – коренът).

Depth	Nodes	Time	Memory
2	1100	.11 seconds	1 megabyte
4	111,100	11 seconds	106 megabytes
6	$10^7$	19 minutes	10 gigabytes
8	$10^9$	31 hours	1 terabytes
10	$10^{11}$	129 days	101 terabytes
12	$10^{13}$	35 years	10 petabytes
14	$10^{15}$	3,523 years	1 exabyte

Time and memory requirements for breadth-firstsearch. The numbers shown assume branching factor  $b = 10$ ; 10,000 nodes/second; 1000 bytes/node.

Фигура 2.

Ако разгледаме таблицата от Фигура 2., ще забележим, че за фиксирана разклоненост  $b=10$  времето и паметта, които са ни необходими, растат експоненциално. Много проблеми, изискващи търсене приемат подобни параметри, когато биват изпълнявани на съвременен компютър. Това ни води до две заключения:

- Първо, необходимата памет е по-голям проблем при търснегото, отколкото времето (31 часа не е толкова голямо притеснение, за търсене на възел от дълбочина 8, но много малко компютри имат 1TB оперативна памет, за да поемат изпълнението на стратегията). Защастие има и други стратегии за търсене, които се нуждаят от много по-малко памет.
- Второто, което можем да заключим, е че изискванията са главен фактор, за всяка стратегия. Проблем от експоненциална сложност, не може да бъде разрешен от стратегия за сляпо търсене, освен ако не е за малки задачи.

### Псевдокод

```

1 procedure BFS( $G, v$ ):
2     create a queue  $\mathcal{Q}$ 
3     enqueue  $v$  onto  $\mathcal{Q}$ 
4     mark  $v$ 
5     while  $\mathcal{Q}$  is not empty:
6          $t \leftarrow \mathcal{Q}.\text{dequeue}()$ 
7         if  $t$  is what we are looking for:
8             return  $t$ 
9         for all edges  $e$  in  $G.\text{adjacentEdges}(t)$  do
12             $u \leftarrow G.\text{adjacentVertex}(t, e)$ 
13            if  $u$  is not marked:
14                mark  $u$ 
15                enqueue  $u$  onto  $\mathcal{Q}$ 

```

### Свойства

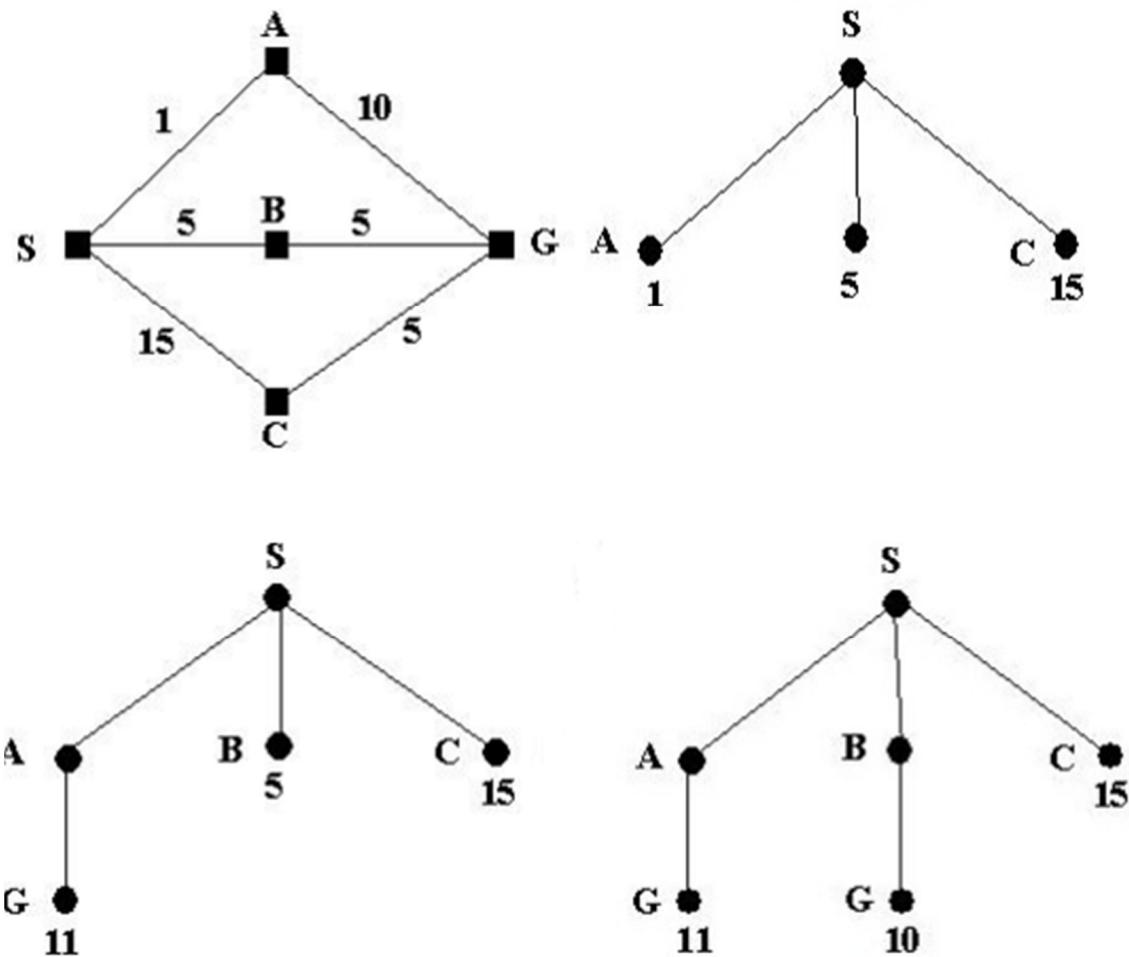
- **-Пълен**: Да (ако  $b$  е крайно)

- **-Време:**  $1 + b + b^2 + b^3 + \dots + bd + bd + 1 - b = O(bd + 1)$
- **-Място:**  $b + b^2 + b^3 + \dots + bd + bd + 1 - b = O(bd + 1)$  (пази всеки възел в паметта)
- **-Оптимален:** Да (ако цената на всяка стъпка е 1)

### 4.3 Търсене с непроменяща се цена (Uniform-cost search)

Търсенето в широчина е оптимално, когато всички стъпки са с равна цена, защото винаги преминава през най-плиткия необходен възел. Посредством просто разширение, ние можем да намерим алгоритъм, който е оптимален за всяка стъпка на ценовата функция. Вместо да обхождаме най-плиткият възел, търсенето с непроменяща се цена разглежда възелът  $n$  с най-ниска цена на пътя. Ако всички възли имат равна цена, Uniform Cost Search ще действа еквивалентно на търсенето в широчина. Алгоритъмът не се интересува от броя стъпки, които има даден път, а само от неговата цена. Поради тази причина може да попадне в безкраен цикъл, ако никога не премине през възел, който има нулева цена, водейки отново до същото състояние. Това условие е също така достатъчно, за да подсигурим, че разходите за пътя се увеличават на всяка стъпка. От това свойство лесно можем да видим, че алгоритъмът се разширява с възли в ред на нарастваща ценова функция. Поради тази причина първият намерен път до търсения възел е оптимален.

## Алгоритъм



Фигура 3.

На Фигура 3. можем да видим как работи алгоритъма. Тъй като търсene с непроменяща се цена се ръководи от ценова функция, а не от дълбочина, няма начин да бъдат лесно оценени b и d. Вместо това, нека  $C^*$  да бъде цената на оптималното решение. Можем да допуснем, че всяко оценяване струва най-малко  $\epsilon$ . Тогава в най-лош случай алгоритъмът ще е със сложност  $O(b^{1+|C^*/\epsilon|})$ , което може да бъде много повече от  $bd$ . Това е така, защото стратегията може да обхожда големи дървета от малки стъпки, преди да търси път, включващ големи и може би полезни стъпки.

## Псевдокод

```

1 procedure UNIFORM-COST-SEARCH (Graph,root,goal):
2     node := root, cost = 0
3     frontier := priority queue containing node only
4     explored := empty set
5     do
6         if frontier is empty
7             return failure
8         node := frontier.pop()

```

```

9     if node is goal
10    return solution
11    explored.add(node)
12    for each of node's neighbors n
13    if n is not in explored
14        if n is not in frontier
15            frontier.add(n)
16    else if n is in frontier with higher cost
17        replace existing node with n

```

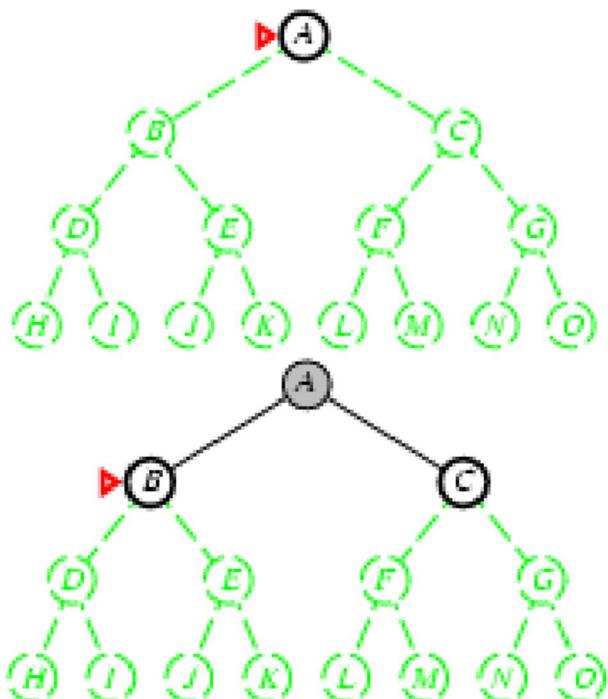
### Свойства

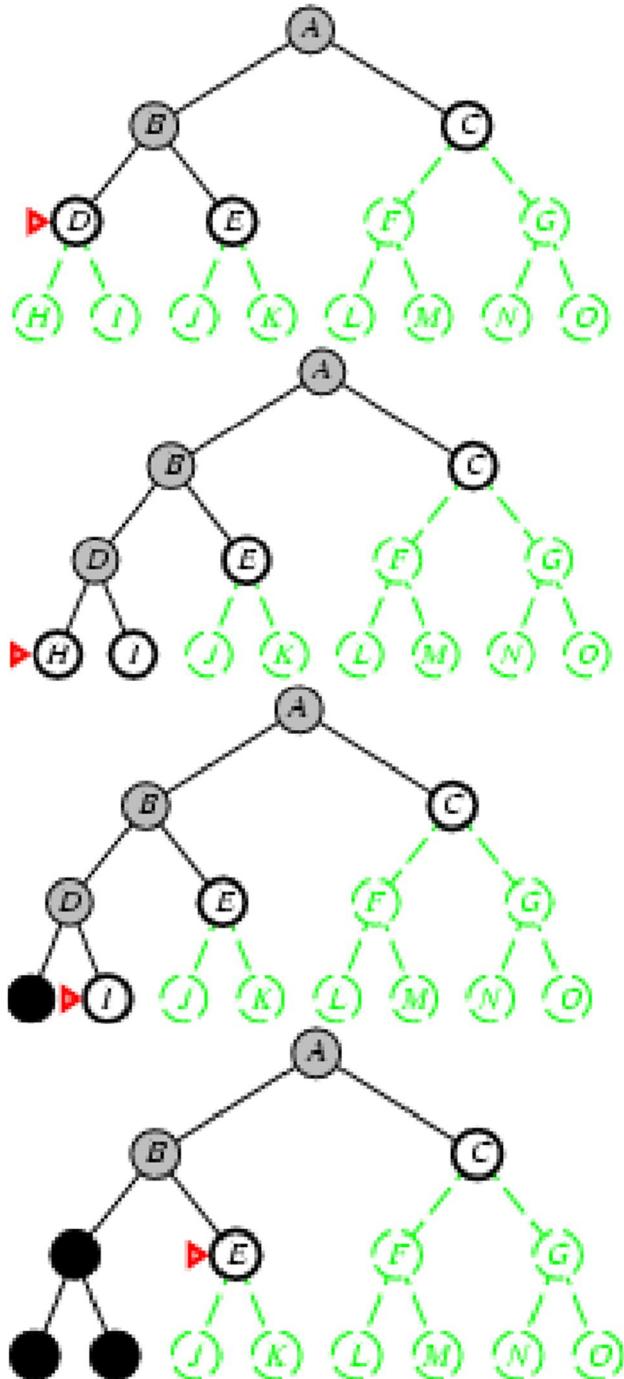
- **Пълен: Да**(ако цената на всяка стъпка е поне  $\epsilon$ )
- **Време:** броят на възли със цена по-малка от оптималното решение,  $O(b^{1+[C^*/\epsilon]})$
- **Място:** броят на възли със цена по-малка от оптималното решение,  $O(b^{1+[C^*/\epsilon]})$
- **Оптимален: Да** (възлите се разширяват спрямо ценова функция)

## 4.4 Търсене в дълбочина (Depth-first search)

Търсенето в дълбочина е стратегия, при която се разширява най-дълбокият възел в сегашната структура, който е необходим. За целта се използва структура от типа LIFO (last-in-first-out), като стак.

### Алгоритъм





Фигура 4.

Както можем да видим във Фигура 4., търсенето в дълбочина винаги разширява най-дълбокият възел в текущата структура. Търсенето продължава моментално към най-дълбокото ниво на търсенето, където възлите нямат наследници. След като бъдат разширени, те се отстраняват от структурата, след което търсенето продължава от най-близкия необходим възел.

Освен с вече споменатата имплементация на алгоритъма чрез стак, търсенето в дълбочина може да бъде имплементирано с рекурсивна функция, която вика себе си за всеки от наследниците си на даден ход. Най-голямото приемущество на търсенето в дълбочина е, че изисква изключително малко памет. Необходимо му е да пази само текущият път от корена,

заедно с останалите необходени наследници на всеки възел от пътя. След като е обходен даден възел, заедно с децата си, той се премахва от паметта. За дадено състояние с разклонение  $b$  и максимална дълбочина  $d$ , алгоритъмът се нуждае от едва  $bm+1$  възела. Използвайки същото предположение като за търснегто в широчина и допускайки, че всички корени, от същата дълбочина като целевия, нямат наследници, виждаме, че стратегията изисква 118KB, вместо 10PB за дълбочина  $d=12$  – около 10 милиарда пъти по-малко пространство.

### Псевдокод

```

1 procedure DEPTH-FIRST-SEARCH( $G, v$ ):
2   label  $v$  as explored
3   for all edges  $e$  in  $G.\text{adjacentEdges}(v)$  do
4     if edge  $e$  is unexplored then
5        $w \leftarrow G.\text{adjacentVertex}(v, e)$ 
6       if vertex  $w$  is unexplored then
7         label  $e$  as a discovery edge
8         recursively call DEPTH-FIRST-SEARCH( $G, w$ )
9       else
10      label  $e$  as a back edge

```

### Свойства

- **-Пълен Не:** проваля се в безкрайно дълбоки пространства, или такива с цикли. Съществуват модификации, които избягват зацикляне. **Пълен е** само в крайни пространства.
- **-Време:**  $O(b^m)$ : ужасно бавно ако  $m$  е много по-голямо от  $d$ . Но ако решението са гъсти, може да бъде много по-бърз от търснегто в широчина
- **-Място:**  $O(bm)$ : линейно време
- **-Оптимален:** Не

## 4.5 Ограничено търсене в дълбочина (Depth-limited search)

### Алгоритъм

Ограничено търсене в дълбочина е алгоритъм, който се справя с проблемите на търснегто на път в безкрайни дървета и може да облекчи стандартното търсене в дълбочина, стигайки само до предварително дефинирано ниво. По този начин всички възли на даденото ниво се третират все едно нямат наследници. За жалост, в следствие на тази техника, възникват нови проблеми. Ако изберем  $m$ , такова, че най-плиткото решение, е на по-дълбоко ниво от  $m$ , алгоритъмът няма да го намери. Също така няма да е оптимален, ако изберем 1 по-голямо  $m$ . Времевата ( $O(b^l)$ ) и пространствената ( $O(bl)$ ) сложност са подобни на тези на търснегто в дълбочина.

В частни случаи ограничено търсене в дълбочина може да бъде базирано на познание върху дадения проблем. Но за повечето проблеми няма как да намерим добро ограничение за

подобно търсене. Тази стратегия може да бъде използвана с малка модификация на стандартния DEPTH-FIRST-SEARCH.

### Псевдокод

```
1 procedure DLS(problem,limit):
2     return RECURSIVE-DLS(MAKE-NODE(Initial-State[problem]), problem, limit)
3
4 procedure RECURSIVE-DLS(node, problem, limit)
5     if Goal-Test[problem](State[node]) then return solution(node)
6     else if Depth[node] = limit then return cutoff
7     else for each successor in Expand(node, problem) do
8         result <- RECURSIVE-DLS(successor, problem, limit)
9         if result = cutoff then cutoff_occurred <- true
10        else if result != failure then return result
11        if cutoff_occurred then return cutoff else return failure
```

### Свойства

- **-Пълен: Не:** възможно е търсеният възел да е на по-дълбоко ниво от предварително дефинираното
- **-Време:  $O(b^l)$ :** където  $l$  е максималното допустимо ниво
- **-Място:  $O(bl)$ :** линейно пространство
- **-Оптимален: Не**

## 4.6 Итеративно търсене в дълбочина

(**Iterative deepening depth-first search**)

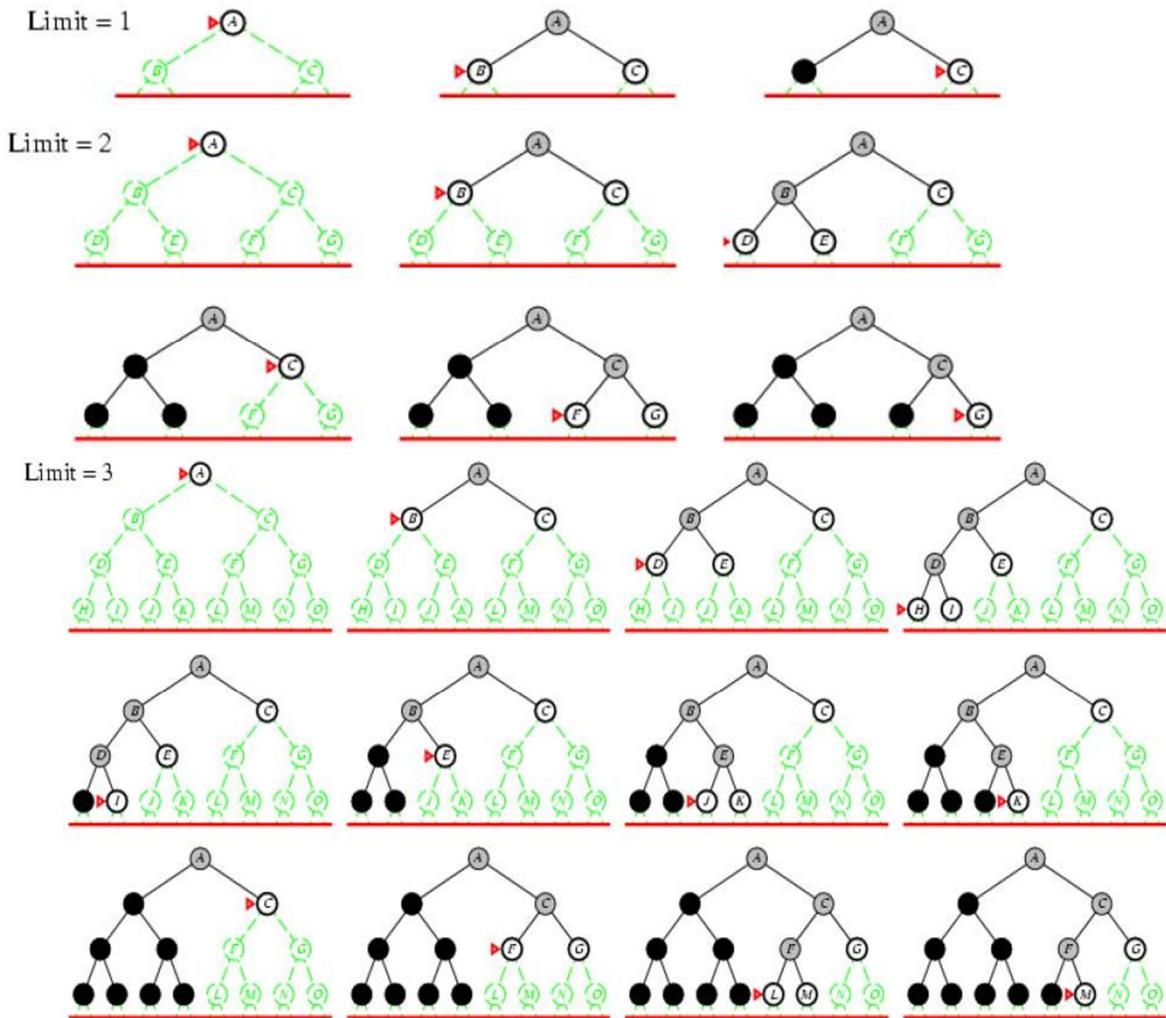
### Алгоритъм

Итеративното търсене в дълбочина е обща стратегия, често използвана в комбинация с търсено в дълбочина. Алгоригмът извършва ограничено търсене в дълбочина, като на всяка стъпка увеличава лимита си на търсене - 0, 1, 2, 3 и т.н.. Завършва, когато стигне до дълбочина  $d$ , на която се намира най-плиткият търсен възел.

По този начин комбинира приемуществата на търсено в дълбочина и търсено в широчина. Подобно на търсено в дълбочина има изключително малки изисквания, и подобно на това в широчина - завършва когато факторът на разколонение е краен. Стратегията е оптимална, когато ценовата функция е ненамаляваща.

Limit = 0





Фигура 5.

Наблюдайки Фигура 5., можем да останем с впечатление, че стратегията е много скъпа във времево отношение, но всъщност се оказва, че не е така. Причината за това е, че дървото на търсене е с почти същия фактор на разклонение и тъй като повечето от възлите са на най-ниското, ниво няма особено значение че генерираме повече от веднъж горните възли. На най-долното текущо ниво ( $d$ ) генерираме веднъж възлите, тези на следващото ниво два пъти и т.н. Така получаваме:

$$(d)b + (d-1)b^2 + \dots + (1)b^d = O(b^d)$$

**В заключение:** итеративното търсене е предпочитаният начин за извършване на сляпо търсене, когато имаме голямо пространство за търсене, и неизвестна дълбочина.

### Псевдокод

```

1. procedure ITERATIVE-DEEPENING-DFS(root, goal)
2.   depth <- 0
3.   repeat
4.     result = DEPTH-LIMITED-SEARCH(root, goal, depth)
5.     if (result is a solution)

```

6. return result
7. depth <- depth + 1

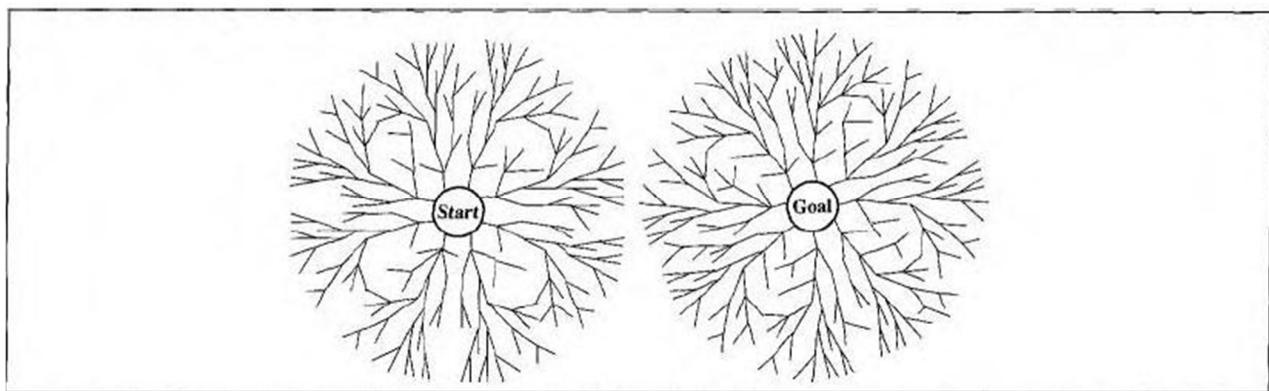
## Свойства

- **-Пълен: Да**
- **-Време:**  $(d+1)b^0 + db^1 + (d-1)b^2 + \dots + b^d = O(b^d)$
- **-Място: O(bd):** линейно време
- **-Оптимален: Да:** ако стъпката е със цена 1

## 4.7 Двупосочнотърсене (Bidirectional search)

### Алгоритъм

Идеята на тази стратегия е да се пуснат успоредно две търсения - едното от коренът, а другото от целта. По този се получава пътят между двете за значително по-малко време. Основната причина да се прави това е, че  $b^{d/2} + b^{d/2}$  е много по-малко от  $b^d$ .



Фигура 6.

(От Фигура 6. можем да забележим, че площта заемана от двата кръга е по-малка от кръг с радиус от началото до целта.)

Огромната разлика в необходимата памет можем да видим ако дадем един прост пример. Ако пуснем търсене в широчина от двете позиции в дърво с разклоненост 10, а търсеният елемент е на дълбочина 3, ще изгенерираме 22,000 възела, което е доста по-малко в сравнение с 11,111,100 необходими при стандартен подход с търсенето в широчина.

Намирането на елементи, които се засичат, не е проблем, т.к. можем да използваме hash таблици и ще отнема константно време, необходимо е само едно от двете дървета да се пазят в паметта и да се проверяват за съвпадения. Това също намалява необходимото физическо пространство -  $O(b^{d/2})$ , но въпреки това то не е никак малко. Друг недостатък на стратегията е, че е неизползваема в ситуации, в които не знаем крайното състояние (примерно търсим позиция за мат в шаха, а има много повече от 1 възможности).

## Свойства

- **-Пълен: Да:** ако се използва търсне в широчина и в двете посоки
- **-Време:**  $O(b^{d/2})$
- **-Място:**  $O(b^{d/2})$
- **-Оптимален: Да:** ако всички стъпки са с равна цена и се използва търсене в широчина

## 4.8 Заключение

Употребата на алгоритми за сляпо търсене е задължителна при решаването на задачи, при които единственото, което ни е известно, е крайното състояние. Доброто познаване на основните методи за търсене ни гарантира леснота и бързина при намирането на решение, както и коректност на резултата. Във Фигура 7. можем да разгледаме и сравним всички споменати до тук алгоритми:

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes <sup>a</sup>	Yes <sup>a,b</sup>	No	No	Yes <sup>a</sup>	Yes <sup>a,d</sup>
Time	$O(b^{d+1})$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^{d+1})$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes <sup>c</sup>	Yes	No	No	Yes <sup>c</sup>	Yes <sup>c,d</sup>

Фигура 7.

Където:

$b$  е факторът на разклонението;

$d$  е дълбочината на най-плиткия търсен възел;

$m$  е максималната дълбочина на дървото;

$\ell$  е лимитът на дълбочината;

<sup>a</sup> - завършен, ако  $b$  е краен;

<sup>b</sup> - завършен, ако цената на стъпката е положително число;

<sup>c</sup> - оптимален, ако всички стъпки са с равна цена;

<sup>d</sup> - ако и двете посоки се използва търсене в широчината;

## 4.9 Методи на 'сляпо' търсене на път до определена цел | Речник

depth	дълбочина
edge	връх/състояние
expand	развитие
explored	обходен
failure	неуспех

goal	цел
goal state	целево състояние
initial state	начално състояние
problem	проблем/задача
recursively	рекурсивно
solution	решение
start state	начално състояние
state	състояние
strategy	стратегия
successor	наследник

## 4.10 Ресурси

Artificial Intelligence: A Modern Approach (2nd Edition) Stuart Russell, Peter Norvig 2002

CS 461 – Artificial Intelligence Pinar Duygulu Bilkent University Spring 2008

[http://en.wikipedia.org/wiki/Breath-first\\_search](http://en.wikipedia.org/wiki/Breath-first_search)  
[http://en.wikipedia.org/wiki/Uniform\\_cost\\_search](http://en.wikipedia.org/wiki/Uniform_cost_search)  
[http://en.wikipedia.org/wiki/Depth-first\\_search](http://en.wikipedia.org/wiki/Depth-first_search)  
[http://en.wikipedia.org/wiki/Depth-limited\\_search](http://en.wikipedia.org/wiki/Depth-limited_search)  
[http://en.wikipedia.org/wiki/Iterative\\_deepening\\_depth-first\\_search](http://en.wikipedia.org/wiki/Iterative_deepening_depth-first_search)  
[http://en.wikipedia.org/wiki/Bidirectional\\_search](http://en.wikipedia.org/wiki/Bidirectional_search)

## **5 Методи на евристичното търсене на път до определена цел.**

В компютърните науки, изкуствения интелект и математическата оптимизация, евристиката е техника, предназначена за решаване на проблема по-бързо, когато класическите методи са твърде бавни, или за намиране на приблизителното решение, когато класическите методи не успяват да намерят точното решение. Това се постига чрез замяна на оптималността, пълнотата и точността със скоростта.

### **Съдържание**

[Съдържание](#)

[Какво е евристично търсене?](#)

[Обща характеристика](#)

[Критерии за използване на евристични функции](#)

[Метод на най-доброто спускане \(Best-First Search\)](#)

[Лакомо търсене \(Greedy Best-First Search\)](#)

[Търсене с минимизиране на общата цена на пътя \(A\\*\)](#)

[Евристично търсене с ограничаване на паметта \(Memory-bounded heuristic search\)](#)

[Рекурсивен метод на най-доброто спускане \(Recursive Best-First Search\)](#)

[Опростен евристичен алгоритъм с ограничаване на паметта \(Simplified memory-bounded A\\*\)](#)

[Евристични функции](#)

[Хипотеза на евристиното търсене\(Heuristic Search Hypothesis\)](#)

[Методи на евристичното търсене на път до определена цел | Речник](#)

[Методи на евристичното търсене на път до определена цел | Ресурси](#)

### **5.1 Какво е евристично търсене?**

Евристичното търсене е техника от методите за търсене на път до определена цел в Изкуствения Интелект, която използва евристична функция за своята реализация. Евристичните функции играят важна роля в стратегията за търсене, заради експоненциалния характер на повечето проблеми. Евристиката помага да се намали броя на възможностите от експоненциално до полиномиално число.

Резултатите за NP-твърдостта в теоретичната информатика правят евристиките единствената жизнеспособна опция за разнообразни сложни оптимизационни задачи, които трябва да бъдат рутинно решени в реални приложения.

Евристичното търсене се възползва от факта, че повечето проблеми предоставят някаква информация, чрез която могат да се разграничават различните състояния по отношение на вероятността да доведат до желаната цел. Тази информация се нарича евристична функция

(Pearl & Korf, 1987). С други думи целта на евристичното търсене е да намали броя на възлите през които се преминава в търсене на целта (Korpec & Marsland, 2001).

Евристичното търсене е информативно търсене „**В което виддаме как информацията за състоянията в пространството може да попречи на алгоритмите да се лутат в мрака!**“ (*Artificial Intelligence a Modern Approach*)

### 5.1.1 Обща характеристика

За разлика от методите на неинформираното (сляпо) търсене, при което намирането на решения на проблемите се свежда до систематично генериране на нови състояния и тяхното сравняване с целевото състояние, което е крайно неефективно в повечето случаи, методите на евристичното търсене (информирано търсене) – тези които използват специфична информация свързана с проблема, която позволява да се конструира евристична функция (оценяваша функция), връщаща небулема оценка (числова оценка в предварителен интервал), могат да намират решения по-ефективно.

Тази оценка може да служи например за мярка на близостта на оценяваното състояние до целта или на необходимия ресурс за достигане от оценяването състояние до целта. Най-често се използва евристична оценяваша функция  $h$ , която връща като резултат приближителната стойност на определен ресурс, необходим за достигане от оценяваното състояние до целта.

Програмната реализация на тези методи се състои в използването на работната памет (спъсък fringe/frontier, открити възли или спъсък на натрупаните/изминати пътища, започващи от началния възел-фронт на търсенето).

### 5.1.2 Критерии за използване на евристични функции

Критерии за вземане на решение дали да се използва евристика за решаването на даден проблем, включват следното:

- Оптималност: Когато съществуват няколко решения за даден проблем, гарантира ли евристиката, че ще бъде намерено най-доброто решение? Действително ли се нуждаем от най-доброто?
- Пълнота: Когато съществуват няколко решения за даден проблем, може ли евристиката да намери всички тях? Имаме ли нужда от всички решения? Много евристики са предназначени само за да се намери едно решение.
- Точност и прецизност: Може ли евристиката да предостави достатъчно точен интервал за предполагаемото решение? Дали границата на грешката на решението е обосновано висока?
- Време за изпълнение: Дали евристиката е най-подходящият метод за решаване на този тип проблем? Някои евристики са много малко по-бързи отколкото класическите методи.

В някои случаи може да е трудно да се разбере дали решението намерено, чрез евристичен подход е достатъчно добро, защото теорията зад евристичните методи не е много сложна. Някои евристични методи са основани на силна теория, те или са извлечени от теорията или

загатнати от експериментални данни. Други са просто правила научени емпирично, без дори намек за теория. Последните са изложени на редица капани.

Когато евристична функция се използва повторно в различни контексти, тъй като се е видяло, че "работи" в един контекст, без да е било математически доказано, че отговаря на определен набор от изисквания, е възможно, че в сегашния набор от данни тя да не доведе до очаквания резултат.

## 5.2 Метод на най-доброто спускане (Best-First Search)

Best-First Search е инстанция на Tree Search или Graph Search алгоритмите в които възелът е оценяваша функция, избран за разширяване основано на оценявашата функция (евристичната)  $f(n)$ . Традиционно се избира възелът с най-малка оценка, защото оценяването измерва разстоянието до целевото състояние. Best-First Search може да се реализира със сортиране на списъка fringe в съответствие с евристичната функция  $f(n)$ . Името Best-First Search е придобило известност, но е неточно, защото ако наистина можехме да разширим най-добрия възел първи, няма да имаме търсене, директно щяхме да стигнем целевия възел. Всичко което можем да направим е да изберем възела, който „изглежда“ най-добър в съответствие с евристичната функция. Ако евристичната функция е точна, това наистина би бил най-добрият възел, в реалността това не е така.

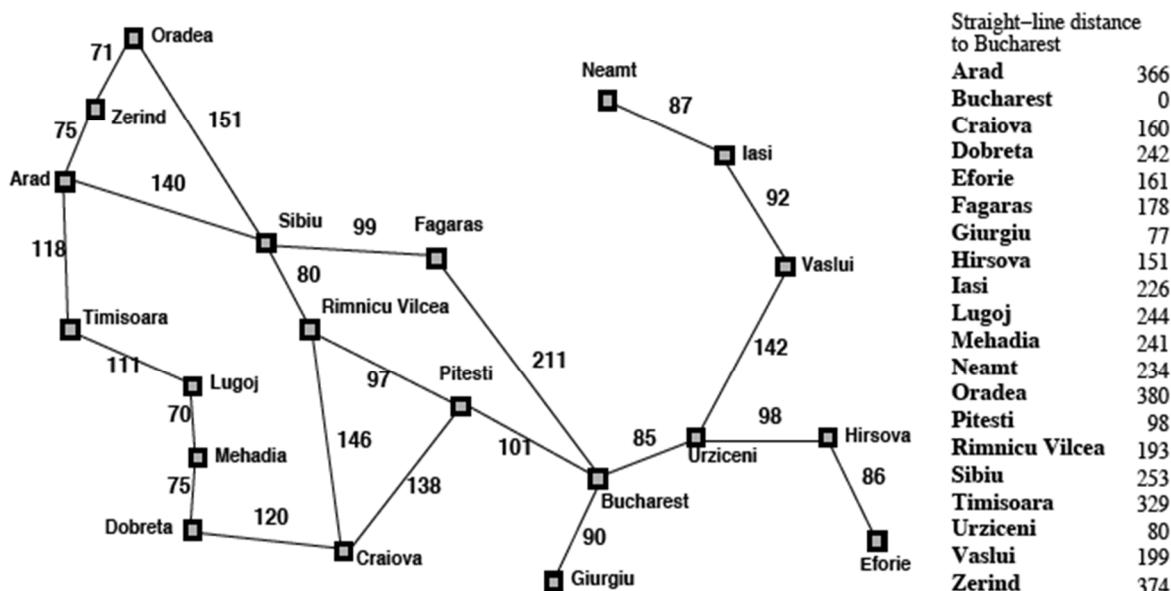
Има цяла фамилия Best-First Search алгоритми с различни оценяващи функции. Ключов компонент на тези алгоритми е евристичната функция, която отбеляваме с  $h(n)$ :

$h(n)$ =оценява приблизително цената на най-евтиния път от възел  $n$  до целевия възел. Ако  $n$  е целевият възел, то  $h(n)=0$ .

Например на картата на Румъния можем да оценим цената на най-евтиния път от Арад до Букурещ посредством разстоянието по права линия между Арад и Букурещ.

Ще използваме данните от тази карта с цел да илюстрираме работата на някои алгоритми за евристично търсене.

## Romania with step costs in km



### Построяване на пътя като списък от последователни състояния

- Фронтът на търсенето е сортиран по отношение на стойностите на евристичната функция списък от пътища (отначало е списък от един път, който включва само началното състояние).
- Ако фронтът има вида  $[p_1, p_2, \dots, p_n]$ , то:
  - Избира се  $p_1$ . Ако  $p_1$  е довел до целта, *край*.
  - Пътищата  $p_{11}, p_{12}, \dots, p_{1k}$ , които разширяват  $p_1$ , се добавят към фронта и новополученият фронт се сортира в нарастващ ред на оценките на пътищата, в резултат на което придобива вида  $[q_1, q_2, \dots, q_{n+k-1}]$ .
  - На следващата стъпка се обработва пътят от фронта, който има най-добра евристична оценка, т.е.  $q_1$ .

### 5.3 Лакомо търсене (Greedy Best-First Search)

Greedy Best – First Search се опитва да разшири възела, който е най-близо до целта с идеята, че това вероятно ще доведе бързо до решение. Така, че той оценява възлите използвайки евристичната функция  $f(n)=h(n)$ . Нека да видим как това работи за намиране на маршрути в Румъния, като целта ни е да стигнем от Арад до Букурещ, използвайки евристика равна на разстоянието по права линия до Букурещ:  $h_{\text{sl}}(n)=$ разстоянието по права линия между  $n$  и Букурещ. Например  $h_{\text{sl}}(\text{Arad})=366$ .

Пример: работа на алгоритъма Greedy Best-First Search при намиране на път върху картата на Румъния.



Фигурата показва прогреса на Greedy Best-First Search използвайки  $h_{\text{sls}}$  да намери пътя между Arad и Bucharest. Първият възел, който трябва да се разшири е от Arad до Sibiu, защото е поблизо до Букурещ и от Zerind и от Timisoara. Следващият възел, който трябва да се разшири ще е Fagaras, защото е най-близо.

### **Свойства на Greedy Best-First Search**

Greedy Best-First Search наподобява Depth-First Search, защото предпочита да следва единичен път до целта, но ще се върне когато стигне задължен край. Също така страда от същите дефекти като Depth-First Search-не е оптимален и не е пълен, защото може да заседне в цикли, в най-лошия случай времевата сложност е  $O(b^m)$ , (където  $m$  е максималната дължочина на търсеният, а  $b$  коефициента на разклоненост на графа на състоянията), но използването на подходяща евристика може да доведе до съществено подобреие. Пространствената сложност на метода е  $O(b^m)$ , тъй като се съхраняват всички достигнати състояния.

### **Псевдокод на Greedy BFS:**

```

1: Procedure: GreedyBFS
2: insert (state=initial_state, h=initial_heuristic, counter=0) into search_queue;
3:
4: while search_queue not empty do
5:   current_queue_entry = pop item from front of search_queue;
6:   current_state = state from current_queue_entry;
7:   current_heuristic = heuristic from current_queue_entry;
8:   starting_counter = counter from current_queue_entry;
9:   applicable_actions = array of actions applicable in current_state;
10:
11:  for all index ?i in applicable_actions  $\geq$  starting_counter do
12:    current_action = applicable_actions[?i];
13:    successor_state = current_state.apply(current_action);
14:
15:    if successor_state is goal then
16:      return plan and exit;
17:    end if
18:    successor_heuristic = heuristic value of successor_state;
19:
20:    if successor_heuristic < current_heuristic then
21:      insert (current_state, current_heuristic, ?i + 1) at front of search_queue;
22:      insert (successor_state, successor_heuristic, 0) at front of search_queue;
23:      break for;
24:
25:    else
26:      insert (successor_state, successor_heuristic, 0) into search_queue;
27:    end if
28:  end for
29: end while
30: exit - no plan found;

```

## 5.4 Търсене с минимизиране на общата цена на пътя (A\*)

Най-широко позната форма на търсене на Best-First Search се нарича търсене A\*, оценява възлите чрез комбиниране на  $g(n)$ -цената да се достигне възела и  $h(n)$ -цената да се достигне от възела до целта.

Евристичната функция има вида:  $f(n) = g(n) + h(n)$ , тоест  $f(n)$  оценява най-евтиния път до целта през  $n$ .

$A^*$  използва допустима/приемлива(admissible) евристика, тоест никога не надценявай цената да стигне до целта. Приемливите евристики са оптимистични, защото приемат, че цената да

се реши проблема е по-малка от колкото в действителност е. Тъй като  $g(n)$  е точната цена до  $n$ , то следва, че  $f(n)$  никога не надценява цената на пътя през  $n$ .

Очевиден пример за такава евристика е  $h_{\text{sls}}$  от горния пример.

### **Построяване на пътя като списък от последователни състояния**

- Фронтът на търсения е сортиран по отношение на стойностите на функцията  $f$  списък от пътища (отначало е списък от един път, който включва само началното състояние).

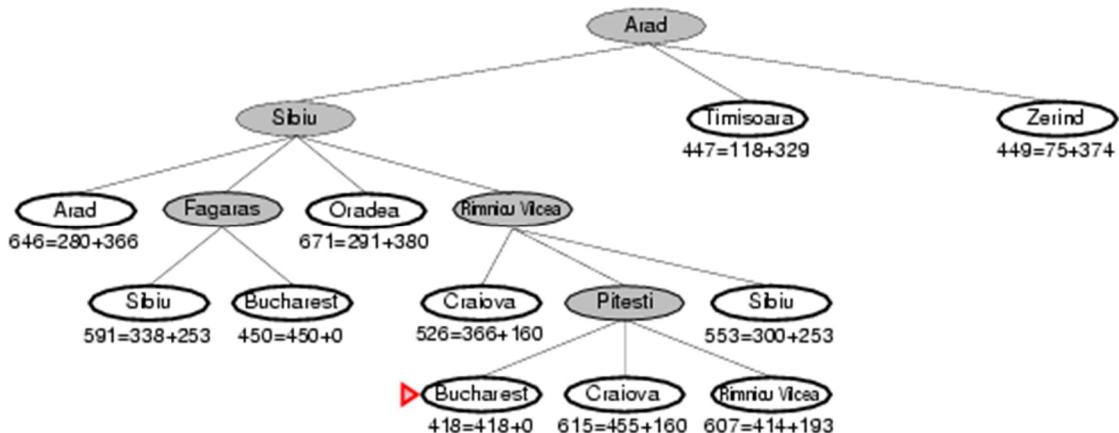
- Ако фронтът има вида  $[p_1, p_2, \dots, p_n]$ , то:

о Избира се  $p_1$ . Ако  $p_1$  е довел до целта, *краий*.

о Пътищата  $p_{11}, p_{12}, \dots, p_{1k}$ , които разширяват  $p_1$ , се добавят към фронта и новополученият фронт се сортира в нарастващ ред на оценките на пътищата, в резултат на което придобива вида  $[q_1, q_2, \dots, q_{n+k-1}]$ .

о На следващата стъпка се обработва пътят от фронта, който има най-добра оценка (най-добра стойност на функцията  $f$ ), т.e.  $q_1$ .

### **Пример: работа на алгоритъма A\* при намиране на път върху картата на Румъния**



**Теорема:** A\* е оптимален.

A\* е оптимален, ако  $h(n)$  е приемлив(admissible).

### **Оптималност на A\*(Стандартно доказателство)**

Да предположим, че никаква неоптимална цел G2 е била генерирана и е в списъка. Нека  $n$  е неразширен възел от най-късия път за оптимална цел G1.

$f(G2) = g(G2)$ , тъй като  $h(G2) = 0$

$> g(G1)$ , тъй като G2 не е оптимално

$\geq f(n)$ , тъй като  $h$  е приемливо

Тъй като  $f(G2) > f(n)$ , A\* никога няма да избере G2 за разширение.

## **Свойства на A\***

Сред оптималните алгоритми от този тип, A\* е оптимално ефективен за всяка евристична функция. Никой друг оптимален алгоритъм не гарантира, че ще разшири по-малко възли от A\*. Това е защото всеки алгоритъм, който не разширява всички възли с  $f(n) < C^*$  рискува да пропусне оптималното решение.

Това че A\* търсенето е пълно, оптимално и оптимално ефективно е много добре, за жалост това не означава, че A\* е отговорът на всички нужди свързани с търсене.

Времевата и пространствена сложност на A\* са експоненциални. Тъй като A\* пази всички генериирани възли в паметта, то обикновенно на A\* не му достига памет, доста преди да има проблеми с времето. Заради това A\* не е практичен за много проблеми с множество данни.

### **Псевдокод на A\*:**

```
function A*(start,goal)
closedset := the empty set // The set of nodes already evaluated.
openset := {start} // The set of tentative nodes to be evaluated, initially containing the start node
came_from := the empty map // The map of navigated nodes.

g_score[start] := 0 // Cost from start along best known path.
// Estimated total cost from start to goal through y.
f_score[start] := g_score[start] + heuristic_cost_estimate(start, goal)

while openset is not empty
    current := the node in openset having the lowest f_score[] value
    if current = goal
        return reconstruct_path(came_from, goal)

    remove current from openset
    add current to closedset
    for each neighbor in neighbor_nodes(current)
        if neighbor in closedset
            continue
        tentative_g_score := g_score[current] + dist_between(current,neighbor)

        if neighbor not in openset or tentative_g_score <= g_score[neighbor]
            came_from[neighbor] := current
            g_score[neighbor] := tentative_g_score
            f_score[neighbor] := g_score[neighbor] + heuristic_cost_estimate(neighbor, goal)
            if neighbor not in openset
                add neighbor to openset

    return failure

function reconstruct_path(came_from, current_node)
if came_from[current_node] in set
```

```

p := reconstruct_path(came_from, came_from[current_node])
return (p + current_node)
else
    return current_node

```

## **5.5 Евристично търсене с ограничаване на паметта (Memory-bounded heuristic search)**

Най-простият начин да се намалят изискванията за паметта на A\* е да се адаптира идеята на итеративното дълбаене(iterative deepening) към евристичното търсене, като резултат на това е алгоритъмът iterative-deepening A\* (IDA\*). Основната разлика между IDA\* и стандартния iterative deepening е в това, че изключването използва цената на  $f(g+h)$ , а не дълбочината; при всяка итерация стойността на изключването е най-малката стойност на  $f$ , която е по-малка от тази стойност в предишната итерация. IDA\* е практичен за много проблеми, като избягва разходите свързани с пазенето на сортираната опашка от възли. За жалост сграда от същите затруднения като Uniform-cost search. Два по-нови алгоритми с ограничаване на паметта са RBFS(Recursive best-first search) и MA\*(memory-bounded A\*).

### **Псевдокод на IDA\*:**

```

algorithm IDA*(start):
loop:
    solution, cost_limit = depth_limited_search([start], cost_limit)
    if solution != None:
        return solution
    if cost_limit == infinity:
        return None

# returns (solution-sequence or None, new cost limit)
algorithm depth_limited_search(path_so_far, cost_limit):
    node = last_element(path_so_far)
    minimum_cost = cost(node)
    if minimum_cost > cost_limit:
        return None, minimum_cost
    if is_goal(node):
        return path_so_far, cost_limit

    next_cost_limit = infinity
    solutions = []
    for s in successors(node):
        new_start_cost = cost(s)
        solution, new_cost_limit = depth_limited_search(extend(path_so_far, s), cost_limit)
        if solution != None:
            solutions.append([solution, new_cost_limit])
        next_cost_limit = min(next_cost_limit, new_cost_limit)

```

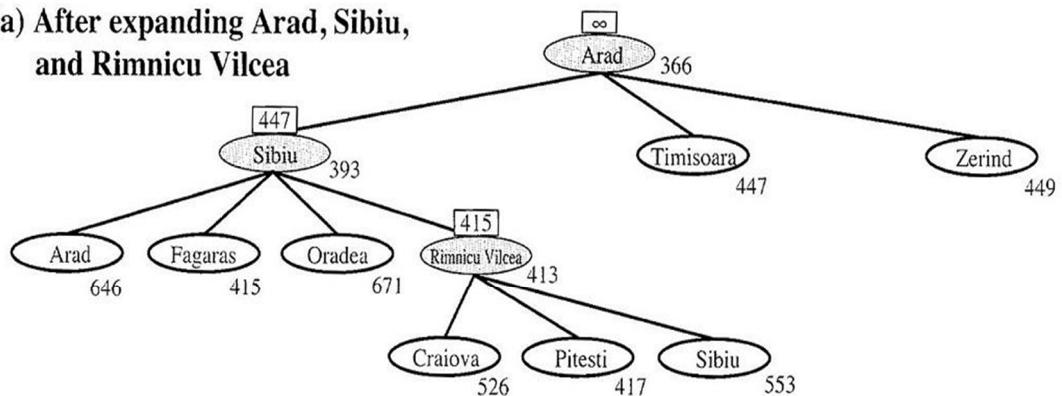
```
if len(solutions) > 0:  
    return lowest_cost_solution_in(solutions)  
return None, next_cost_limit
```

## 5.6 Рекурсивен метод на най-доброто спускане (Recursive Best-First Search)

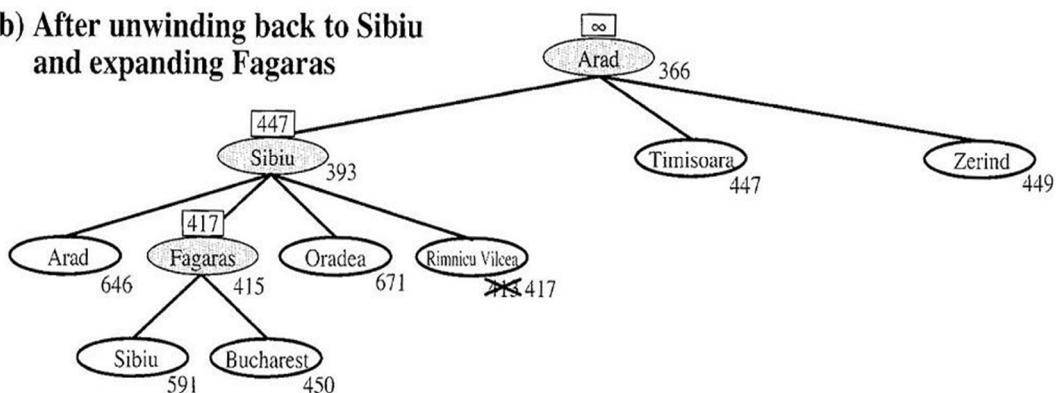
Това е прост рекурсивен алгоритъм, който се опитва да наподобява стандартния Best-First Search, но използвайки само линейна сложност по памет. Неговата структура е подобна на Recursive Depth-First Search, но вместо да продължава неопределено по настоящия път, пази запис за стойността на  $f$  на най-добрния алтернативен път, възможен да се поеме от всеки предшественик на настоящия възел. С развиването на рекурсията, RBFS заменя стойността на  $f$  за всеки възел по пътя с най-добрата стойност на  $f$  на неговите деца. По този начин RBFS помни стойността на  $f$  на най-доброто листо на преминатото поддърво и може да реши дали си заслужава повторното разширяване на това поддърво на по-късен етап.

Фигурата показва как чрез RBFS се достига Букурещ.

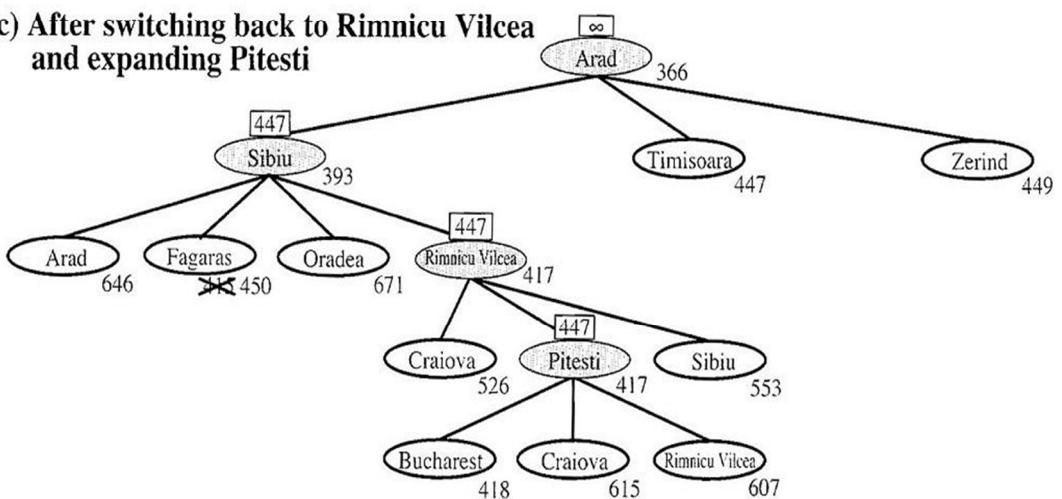
(a) After expanding Arad, Sibiu, and Rimnicu Vilcea



(b) After unwinding back to Sibiu and expanding Fagaras



(c) After switching back to Rimnicu Vilcea and expanding Pitesti



## **Свойства на RBFS**

Подобно на A\*, RBFS е оптимален алгоритъм, ако евристичната функция  $h(n)$  е приемлива. Неговата сложност по памет е линейна спрямо дълбината на най-дълбокото оптимално решение, но сложността по време е трудно да бъде характеризирана: тя зависи както от точността на евристичната функция, така и от колко често най-добрият път се променя, докато възлите се разширяват.

Както IDA\* така и RBFS страдат от това, че използват малко паметта. Между итерациите IDA\* помни само едно число: настоящата цена на  $f$ . RBFS пази повече информация в паметта, но използва само линейно паметта, тоест дори да имаме повече памет, той няма да се възползва от нея.

Логично е да използваме цялата предоставена памет. Два алгоритъма който правят това са MA\* (memory-bounded A\*) и SMA\*(simplified MA\*). Ще разгледаме SMA\*, който е по-прост.

## **5.7 Опростен евристичен алгоритъм с ограничаване на паметта (Simplified memory-bounded A\*)**

SMA\* процедира както A\*, разширява най-доброто листо докато паметта се напълни. На този момент не може да се добавя нов възел към търсенето без да се махне стар. SMA\* винаги маха най-лошия възел-този с най-голяма стойност на  $f$ . Както RBFS, SMA\* запазва стойността на забравения възел на възела на родителя му. По този начин предшественикът на забравеното поддърво знае стойността на най-добрния път на това поддърво. С тази информация, SMA\* регенерира това поддърво само когато всички други пътища се оказват, че изглеждат по-неприемливи отколкото забравеният път.

Казахме, че SMA\* разширява най-добрия възел и изтрива най-лошия. Какво ще се случи ако всички възли имат еднакви стойности на  $f$ ? Тогава алгоритъмът може да избере един и същ възел и за двете действия. SMA\* решава този проблем като разширява най-новия най-добър възел и изтрива най-стария най-лош възел.

### **Свойства на SMA\***

SMA\* е пълен, ако има достъпимо решение- това е така ако дълбината на най-плиткия целеви възел е по-малък от размера на паметта. Оптимален е ако е достъпимо някакво оптимално решение; в противен случай връща най-доброто достъпимо решение. В практиката SMA\* може спокойно да бъде най-добрият алгоритъм за общи цели за намиране на оптимални решения, особено когато пространството на състоянията е граф и генерацията от възли е скъпа в сравнение с допълнителните разходи свързани с поддържането на отворени и закрити списъци.

### **Псевдокод на SMA\*:**

```
function SMA-star(problem): path
    queue: set of nodes, ordered by f-cost;
    begin
        queue.insert(problem.root-node);
```

```

while True do begin
if queue.empty() then return failure; //there is no solution that fits in the given memory
node := queue.begin(); // min-f-cost-node
if problem.is-goal(node) then return success;

s := next-successor(node)
if !problem.is-goal(s) && depth(s) == max_depth then
f(s) := inf;
// there is no memory left to go past s, so the entire path is useless
else
f(s) := max(f(node), g(s) + h(s));
// f-value of the successor is the maximum of
// f-value of the parent and
// heuristic of the successor + path length to the successor
endif
if no more successors then
update node-s f-cost and those of its ancestors if needed

if node.successors ⊆ queue then queue.remove(node);
// all children have already been added to the queue via a shorter way
if memory is full then begin
badNode := shallowest node with highest f-cost;
for parent in badNode.parents do begin
parent.successors.remove(badNode);
if needed then queue.insert(parent);
endfor
endif

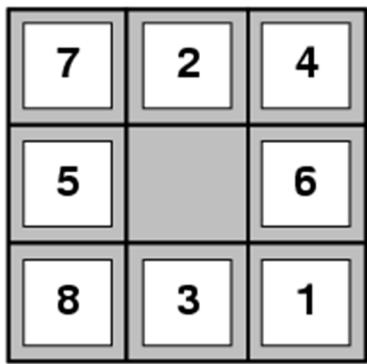
queue.insert(s);
endwhile
end

```

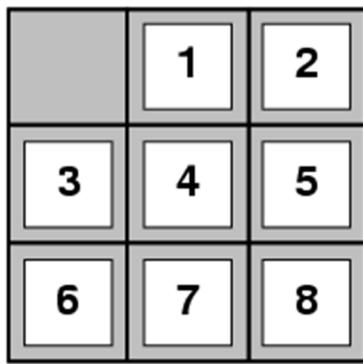
## 5.8 Евристични функции

Ще разгледаме евристики свързани със задачата за 8-те плъзгащи се плошки, с цел да разгледаме евристичните функции като цяло.

Задачата за 8-те плъзгащи се плошки е една от най-ранните проблеми свързание с алгоритми за търсене. Задачата на пъзела е да се плъзгат плоцките вертикално или хоризонтално като се използва празното пространство докато се достигне целевото състояние. Средната цена на решението за случаен генериран начин състояния е около 22 стъпки. Коефициентът на разклонение е 3. Това означава, че изчерпателно търсене с дълбочина 22 ще е около  $3^{22}$ , но за 15-плошков пъзел е около  $10^{13}$ , заради което трябва да се намери добра евристика.



Start State



Goal State

Ако искаме да открием най-късия път използвайки A\*, ние необходима евристична функция, която никога не надценява броя на стъпките до целта. Имаме две най-често използвани евристични функции:

- $h_1$  = броя на плочките, чиято текуща позиция е различна от позицията им в целевото състояние
- $h_2$  = товаалното (сумарното) Манхатънско разстояние

Манхатънско разстояние (Manhattan Distance) между точките  $(X_i, Y_i)$  и  $(X_j, Y_j)$ :  $D = |X_i - X_j| + |Y_i - Y_j|$ .

Пример:

$$h_1(\text{Start}) = 8$$

$$h_2(\text{Start}) = 3+1+2+2+2+3+3+2 = 18$$

## 5.9 Хипотеза на евристичното търсене (Heuristic Search Hypothesis)

В своята реч по случай награждаването им с наградата Turing, Allen Newell и Herbert A. Simon обсъждат евристичната хипотеза: физична символна система (physical symbol system) която постоянно ще генерира и модифицира познати символни структури докато генерираната структура не съвпадне със структурата на решението. Всяка следваща итерация зависи от стъпката преди нея, като по този начин евристичното търсене научава какви възможности да преследва и кои да се пренебрегнат измервайки, колко близо е настоящата итерация до целта. Следователно, никога няма да бъдат генериирани някои възможности, тъй като те са измерени, че имат по-малка вероятност да доведат до решение.

Евристичният метод може да изпълни тази задача с помощта на дървета за търсене.

## 5.10 Речник

<i>Heuristic</i>	Евристика
<i>Heuristic function</i>	Евристична функция (оценяваша функция)
<i>Optimality</i>	Оптималност

<i>Completeness</i>	Пълнота
<i>Time Complexity</i>	Сложност по време
<i>Space Complexity</i>	Сложност по памет(Пространствена)
<i>State space</i>	Пространство на състоянията
<i>Node</i>	Възел
<i>Evaluation Function</i>	Оценяваша функция
<i>Expansion</i>	Разширяване
<i>Goal state</i>	Целево състояние
<i>Admissible</i>	Приемлива(оптимистичен)

## 5.11 Ресурси

Artificial Intelligence: A Modern Approach (2nd Edition) Stuart Russell, Peter Norvig 2002	Artificial Intelligence Lections Maria Nisheva
<a href="http://en.wikipedia.org">en.wikipedia.org</a>	<a href="http://www.google.com">www.google.com</a>

## **6 Локално търсещи алгоритми. Генетични алгоритми**

Алгоритмите за локално търсене представляват мета-евристичен подход за намиране на решения на трудно изчислими оптимизационни задачи. Те намират широка употреба при редица трудни изчислителни проблеми, включително и проблеми от компютърните науки (най-вече Изкуствен Интелект), математика, изследване на операциите, инженерство и биоинформатика.

Генетичните алгоритми са специален вид евристични алгоритми, наподобяващи биологични процеси. Те генерираят решения на оптимизационни проблеми използвайки техники вдъхновени от естествената еволюция, като *наследяване, мутация, селекция, кръстосване*.

### **Съдържание:**

[Какво представляват локално търсещите алгоритми?](#)

[Запоз локално търсещи алгоритми?](#)

[Видове локално търсещи алгоритми](#)

[Лакоми. Катерене на хълм \(Hill-climbing\)](#)

[Дефиниция](#)

[Псевдо-код](#)

[Проблеми](#)

[Преодоляване на проблемите](#)

[Рандомизирани](#)

[WalkSAT](#)

[Симулирано каляване \(Simulated annealing\)](#)

[Генетични алгоритми](#)

[Основни понятия](#)

[Хромозоми](#)

[Селекция](#)

[Популация](#)

[Кръстосване](#)

[Мутация](#)

[Ход на алгоритма](#)

[Инициализация](#)

[Селекция](#)

[Генетични операции](#)

[Приключване на процеса](#)

[Речник](#)

## **6.1 Какво представляват локално търсещите алгоритми?**

Локалното търсене е непълен способ за намиране на решение на даден оптимизационен проблем. Алгоритъмът стартира раздавайки стойности на всичките променливи, след което се движи из набора от възможни решения, прилагайки локални промени итеративно до постигане на крайната цел (намиране на достатъчно добро решение, достигане на лимит от време или брой итерации и т.н.). Локално търсене може да бъде ползвано върху проблеми, които могат да бъдат формулирани като „максимизиране на печалба по определен критерий измежду краен набор от възможни решения“. Обикновено, тези алгоритми изменят стойностите на променлива на всяка стъпка. Всяка следваща стойност е близка до предишната, от където идва и името им - „Локално търсещи алгоритми“. Те използват функция която оценява качеството на текущото състояние (например брой нарушени изисквания). Тази „оценка“ се нарича цена и целта на алгоритъма е да намери решението, което я свежда до минимум.

### **Защо локално търсещи алгоритми?**

За да намерим най-добро решение на даден проблем, ние трябва да сравним всички възможни решения. Често наборът от възможните решения е толкова голям, че за разумен период от време не можем да обходим цялото пространство на решението и да получим смислени резултати. Локално търсещите алгоритми са създадени така, че да не претърсват цялото пространство на решението и да работят бързо за много големи набори от възможни решения. В средния случай те намират сравнително добри решения за относително кратко време. Те не гарантират, че могат да намерят оптимално решение, тоест не могат да докажат, че решение не съществува. Използват се предимно в случаи, в които се знае, че решенията съществуват със сигурност (или е много вероятно).

### **Видове локално търсещи алгоритми**

Има 2 основни класа локално търсещи алгоритми:

Лакоми (greedy) или не-рандомизирани (non-randomized). Тези алгоритми се опитват подобряват цената (или да не я влошат) на всяка следваща стъпка. Основният им е проблем е наличието на „плата“ - място, на което няма локални ходове, които могат да подобрят текущото състояние.

Другият клас са рандомизирани, алгоритми, създадени специално за да решат проблема на лакомите алгоритми. За да избягат от плата, тези алгоритми предприемат случайни стъпки по време на процеса.

## **6.2 Катерене по хълм (Hill-climbing)**

Итеративен алгоритъм, който стартира произволно решение и се опитва да намери по-добро изменяйки елемент от решението инкрементално. Ако промяната даде по-добър резултат, се извършва нова промяна на база на текущата. Алгоритъмът спира, когато не може да направи промяна, която да подобри текущото решение. Този алгоритъм може да бъде приложен към проблем с [търговския пътник](#). Алгоритъмът стартира с никакво първоначално решение,

обхождащо всички градове като постепенно го подобрява, например сменяйки реда, в който дава града ще бъдат обходени.

Катеренето по хълм е добър метод за намиране на локални оптимуми, но не гарантира най-добро решение. Относителната му простота го правят доста популярен избор между оптимизационните алгоритми. Намира широко приложение в Изкуствения интелект. Макар да има други алгоритми, които дават по-точно резултати, в някои ситуации катеренето по хълм може да се представя не по-зле от тях. То е особено полезно, когато основното ограничение идва не от прецизността, а от времето, за което трябва да бъде намерено



решение. Алгоритъмът може да даде валидно решение, дори да бъде прекъснат в процеса на търсене.

#### Псевдо-код:

```

function Hill-Climbing(problem) returns a state that is a local maximum
    inputs: problem, a problem
            local variables: current, a node
            neighbor, a node
    current  $\leftarrow$  Make-Node(Initial-State[problem])
    loop do
        neighbor  $\leftarrow$  a lowest-valued successor of current
        if Value[neighbor]  $\geq$  Value[current]
            then return State[current]
        current  $\leftarrow$  neighbor
    end

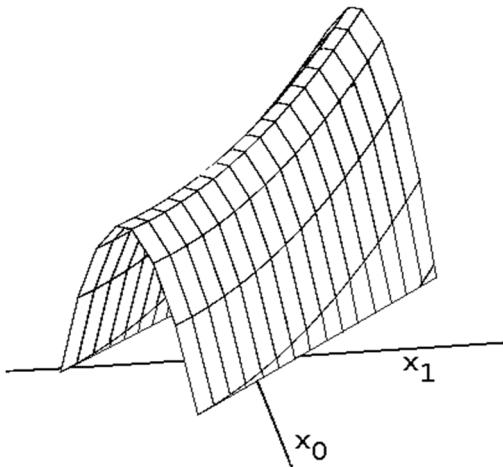
```

На всяка стъпка отиваме към съсед, който има който има по-ниска стойност. Лесно може да бъде модифициран за максимизиране.

#### Проблеми на алгоритъма:

**Локален максимум** - освен ако пространството на състояниата няма изпъкната повърхност, този алгоритъм в повечето случаи ще намира локални решения.

**Хребети** - те представляват голямо предизвикателство за този тип алгоритми, тъй като те променят само по 1 променлива на стъпка и промените се движат винаги по осите. Това означава, че алгоритъмът може да прави малки стъпки зигзагообразно в опита си да стигне до оптимум, а това може да бъде прекалено времеотнемащо.



Source: [://en.wikipedia.org/wiki/File:Ridge.png](https://en.wikipedia.org/wiki/File:Ridge.png)

**Плата** - плато се получава, когато пространството на решенията е достатъчно плоско, че оценъчната функция да не може да различи стойностите. В такъв случай алгоритъмът може да няма възможността да определи на къде да продължи своето катерене и да се лута в посоки, които не водят до решение.

#### Преодоляване на ограниченията с локалните оптимуми.

За да бъдат решени някои специфични проблеми, има различни разновидности на базовия алгоритъм, които предлагат различни предимства.

**Просто катерене по хълм** - избира се първият по-добър съсед. Страда от всички проблеми.

**Най-стръмно изкачване** - сравняват се всички съседи и се избира този, който има най-добра оценка. Подобен на най-добро първо търсене ([best-first search](#)). Страда от всички проблеми.

**Стохастично катерене по хълм** - изследва случаен съсед и според оценката му решава дали да продължи по него или да изследва друг съсед.

**Катерене по хълм със случайно рестартиране** (Random-restart hill climbing) - метаалгоритъм, който в основата си представлява катерене по хълм със случайни начални състояния. Крайните резултати от изследванията се сравняват помежду си и се избира най-доброто. Този алгоритъм е учудващо ефективен и се оказа, че доста често, е по-добре да отделиш процесорно време за случаини изследвания на пространството от решенията от колкото за внимателни подобрения на началното състояние.

Претегляне на ограниченията или метод на пробива - използва претеглена сума на нарушените ограничения, като мярка за цената. Гледа да промени това тегло, когато няма ход, който внася подобрения. Т.е. ако няма текущ ход, който да подобрява стойността на решението, алгоритъмът увеличава теглото на ограниченията, които са нарушени. По този начин, всеки ход, който иначе, не би променил цената, я намаля. Цената на ограниченията, които биват нарушавани в няколко последователни хода, расте непрестанно.

**„Табу” търсене** - проблемът на катеренето по хълм, което позволява обхождането на съседи, които не намалят цената е, че може да зацикли. „Табу” търсенето се справя с този проблем като пази лист от „забранени” промени. В общия случай, този лист съдържа скорошните промени (двойки променлива - стойност) и не позволява те да се повторят. Списъкът се опреснява всеки път, когато има промяна в текущото състояние. Ако на променлива бъде даде стойност, то двойката променлива-стойност се записва в списъка и най-старата такава двойка се изкарва от него. Листът съдържа само ограничен брой скорошни променливи. Алгоритъмът избира само най-добрия ход измежду тези, който не са в табу листа. По този начин, зациклияне може да се получи, само ако табу листът е по-кратък от комбинацията от съседи с еднакви стойности.

### **6.3 Рандомизирани алгоритми (randomized).**

#### **6.3.1 Случайна разходка (Random walk).**

Този тип алгоритми се движат понякога като лакоми, но понякога вземат случайни решения, които не се съобразяват с оценъчната функция. Факторът на случайност се определя от параметър.

**WalkSAT.** Алгоритъм за разрешаване на проблеми за удоволетворение на булеви изрази. Работи върху изрази в конюнктивна нормална форма. Алгоритъмът започва работа като присвоява случайни стойности на всяка променлива. Ако състоянието удоволетворява всички клаузи, алгоритъмът приключва работа. В противен случай, някоя от променливите сменя стойността си и горното се повтаря докато всички клаузи не са удоволетворени. Алгоритъмът избира клауза, която е неудоволетворена в текущото състояние и сменя стойност на променлива в тази клауза. В общия случай измежду всички неудоволетворени клаузи, се избира някоя на случаен принцип. Променливата се избира така, че най-малък брой предно удоволетворени клаузи да станат неудоволетворени, с някаква вероятност за избиране на чисто случайна променлива. Когато се избира на случаен принцип, се гарантира с шанс ( $1 / \text{брой променливи в клаузата}$ ), че ходът ще поправи текущо некоректно разпределение. Алгоритъмът може да рестартира с ново случайно разпределение, ако решение не бъде намерено за определен период от време, като способ за избягване на локални минимуми за броя на неудоволетворени клаузи. Съществуват много и различни версии на WalkSAT. Алгоритъмът се е доказал като особено ефективен за намирането на решения на проблеми образувани при конверсията от проблеми за автоматично планиране. Подходът към планирането, който конвертира проблема с планирането в проблем за булева удоволетвореност се нарича **satplan**.

MaxWalkSAT е вариант на WalkSAT, проектиран да разрешава проблем със удоволетвореност, при който клаузите имат различна цена и крайната цел е да се намери такова разпределение, че сумата от теглата на нарушените клаузи да е минимално (пълно решение може и да не се изисква).

#### **6.3.2 Симулирано каляване (СК) (Simulated annealing).**

Представлява обща мета-евристика, която се опитва да се справи с намирането на добра апроксимация на глобалния максимум на дадена функция в големи пространства с възможни

решения. Често се ползва в дискретни пространства. Името и идеята са вдъхновени от закаляването в металургията, техника включваща нагряване и контролирано изстиване на метала, с цел да се повиши размера на кристалите и да се намалят техните дефекти (и двете свойства зависят от термодинамичната енергия). Именно идеята за плавното охлаждане се ползва в този алгоритъм като постепенно се намаля възможността за приемане на лоши решения по време на изследването на пространството. Предприемането на „лоши“ ходове в ключова част при мета-евристичните подходи, тъй като увеличава шансовете за намиране на решение, по-близко до глобалния оптимум.

**Общ преглед:** При този метод, всяка точка  $S$  от пространството е аналогична на някакво състояние на физична система и функцията  $E(s)$ , която трябва да бъде минимизирана е аналогът на вътрешната енергия на системата в началното състояние. Целта е да се доведе системата до състояние с възможно най-малко вътрешна енергия.

**Основната итерация:** На всяка стъпка, СК разглежда някое съседно състояние спрямо текущото и решава вероятностно дали да продължи към него или да остане в текущото. В крайна сметка, тези вероятности, водят системата към състояние с по-ниска енергия. Обикновено стъпката се повтаря докато системата стигне състояние, което е достатъчно добро или определени предварително зададени граници са подминати.

Съседите: Съседите на едно състояние са нови състояния на проблема, които се получават чрез промяна на текущото по определен метод. Например при проблема с търговския пътник, всяко състояние типично се дефинира като пермутация на градовете, които трябва да бъдат посетени. Съседите са друга пермутация, която е получена, например, чрез размяната на двойка съседни градове. Действието, което се предприема с цел да се намери друго решение се нарича „ход“ и различните ходове дават различни съседи. Тези ходове обикновено водят до минимални разлики в решението и е добре да помогнем на алгоритъма да оптимизира решението максимално, като в същото време поддържа досега намерените оптимални части и променя само суб-оптималните. За проблема с локалните оптимуми, мета-евристичните подходи допускат предприемането на „лоши“ стъпки. Това означава, че алгоритъмът няма да спре на локалния оптимум, а ще продължи да търси и след безкрайно дълго време би трябвало да намери глобалния оптимум.

### Вероятности на одобрение

Вероятността да се направи преход от текущото състояние  $s$  към някое съседно  $s'$  се определя от вероятностна функция  $P(e, e', T)$ , която зависи от енергиите на двете състояния  $e = E(s)$ ,  $e' = E(s')$  и от глобален, вариращ с времето параметър  $T$ , наречен „температура“. Състояние с по-малко енергия е по-добре от състояние с по-висока енергия. Вероятностната функция  $P$  трябва да бъде положителна, дори когато  $e' > e$ . Това предпазва метода от забиване в локален минимум. Когато  $T$  клони към нула, вероятността  $P(e, e', T)$  трябва да клони към нула, ако  $e' > e$ . С намалянето на стойностите на  $T$ , алгоритъмът ще започва да предпочита все повече по-полезни ходове и все по-малко тези, които имат отрицателна полза. При  $T = 0$ , алгоритъмът става обикновен „лаком“.

### Псевдокод:

```
s ← s0; e ← E(s)      // Начално състояние, енергия.  
sbest ← s; ebest ← e // Начално най-добро състояние
```

```

k ← 0           // Брой оценки.
while k < kmax and e > emax // Докато има време и не е достатъчно добро:
    T ← temperature(k/kmax)    // Изчисление на температурата.
    snew ← neighbour(s)      // Избор на съсед.
    enew ← E(snew) // Оценяване на енергията.
    if P(e, enew, T) > random() then // Да се преместим ли?
        s ← snew; e ← enew // Да.
        if enew < ebest then      // Нов оптимум?
            sbest ← snew; ebest ← enew // Запазване на новия съсед като най-добър.
        k ← k + 1 // Увеличаване на броя на оценките
    return sbest

```

### 6.3.3 Генетични алгоритми

Генетичните алгоритми1 (ГА) са стохастичен метод за глобално търсене и оптимизация, който имитира еволюцията на живите индивиди, описана от Чарлз Дарвин в “За произхода на видовете и значението на естественият подбор”.

В еволюционните алгоритми се използват трите основни принципа на естествената еволюция, описани от Дарвин: репродукция, естествен подбор и разнообразие на индивидите, поддържано чрез разликите на всяко поколение с предишното.

#### Основни понятия

**Хромозоми.** При генетичните алгоритми хромозомите представляват набор от гени, които кодират неизвестните променливи. Всяка хромозома представлява допустимо решение на поставената задача. Индивид е вектор на променливите ще бъдат използвани като понятия еквивалентни на хромозома. Гените от своя страна могат да бъдат булеви, целочисленi, с плаваща запетая и символни низове, както и всяка тяхна комбинация

Тип	Пример
булев	[0, 0, 0, 1, 0, 1, 1, 0]
Целочислен	[ 12, 123, 654, -123]
плаваща запетая	[3.5, 2.7, -5.231, 5]
символни низове	["abx", "qwe", "rte"]
комбинация	["abx", 123, 0.14]

**Селекция.** При еволюционните алгоритми селекцията на най-добрите индивиди става въз основа на функция на приспособимост, даваща оценка на конкретния индивид. Например такава функция може да е квадратичен критерий на грешката между изходите на желана от нас система и реалната, близост на полюсите на затворена система до желаните и т.н. Ако задачата е за минимизация, то индивидите с по-малка стойност на функционала ще имат по-голям шанс да бъдат избрани за рекомбинация и съответно за продължаване на поколението.

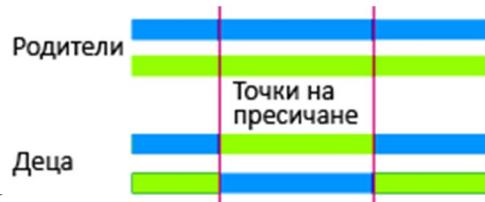
**Популация.** Наборът от всички текущи индивиди върху, които работим се нарича популация.

**Кръстосване (Рекомбинация),** При репродукцията, първо се появява рекомбинацията (кръстосване или кросинговър). При нея гените от родителите формират по някакъв начин изцяло нова хромозома. Типичната рекомбинация при генетичните алгоритми е операция, изискваща два (възможно и повече) родителя.

**Едноточково кръстосване** - избира се точка на пресичане (една и съща за двамата родители) и се сменят съответните части от хромозомите.



**Двуточково пресичане** - избират се 2 точки на пресичане (еднакви за двамата родители) и се

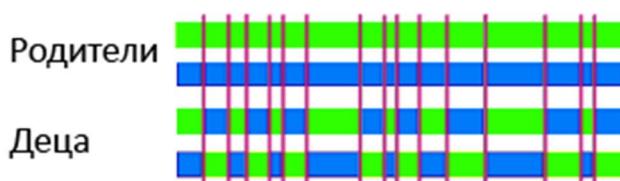


сменят парчетата между двете точки.

**Рязане и снаждане** - избират се различни точки при различните родители и се сменят парчетата след точките. Децата може да са с различна дължина.



**Еднообразно кръстосване** - използва фиксиран отношение на миксиране между родителите. За разлика от по-горните примери, в този родителите допринасят повече на генно ниво от колкото на сегментно. С коефициент 0.5, поколението има приблизително равен дял от гените на родителите си, въпреки че точките на пресичане могат да бъдат избрани на случаен



принцип.

**Кръстосване на трима родители** - детето произлиза от трима случајно избрани родители. По-често използвано при булеви хромозоми. Всеки бит от първият родител се сравнява със съответствания му бит от втория родител. Ако съвпадат, този бит се присвоява на наследника, ако не се взима бита от третия родител. Пример

1	1	0	1	0	0	0	1	0
+								

0	1	1	0	0	1	0	0	1
+								
1	1	0	1	1	0	1	0	1
=								
<b>1 1 0 1 0 0 0 0 1</b>								

**Кръстосване на наредени хромозоми** - в зависимост от това как хромозомите представят решението, не винаги директно разместване на части от хромозомите може да доведе до получаването на валидно състояние. Пример за това е, когато хромозомата представява нареден списък, като например списък с градовете през, които ще премине търговския пътник от небезизвестната задача. Има много методи за пресичане на наредени хромозоми. Горните методи могат да бъдат използвани за целта, но ще се наложи известна „поправка”.

Ето някои методи, които могат да бъдат използвани в случая:

1. **Частично-съчетано кръстосване (partially matched crossover)**- в този метод 2 точки на пресичане се избират на случаен принцип. Двете точки задават селекция(mapping). Кръстосването става, като по създадения мапинг се разменят елементите.

Пример:

1 2 | 3 4 | 5 родител1 оформеният мапинг е ([3, 4], [4, 5]) при точки на пресичане **2 и 4**

1 3 | 4 5 | 2 родител2 извършват се сменни според мапинга

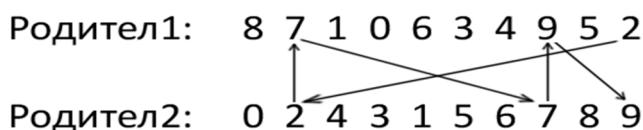
1 4 | 3 5 | 2      стъпка 1

1 5 | 3 4 | 2      стъпка 2

1 5 3 4 2      резултат

2. **Циклично кръстосване (cycle crossover)** - Започва се от някой ген **K** в първия родител, **K**-ят ген във вторият родител се сменя с него и процедурата се повтаря за изместеният ген, докато не се замени първият хванат (не се затвори цикъла).

Пример:



Цикъл: 8 2 0 3 7 5 9

**Мутация.** Новосъздаденото чрез селекция и кръстосване потомство след това може да бъде подложено на мутация. Мутация означава, че елементи от ДНК се променят. Тези промени са породени главно от грешки при копирането на гените от родителите. В термините на генетичните алгоритми мутация означава произволна промяна на стойността на ген в поколението. Хромозомата, чийто ген ще бъде променен, и мястото на гена също се избират по случаен принцип. Целта на мутацията е да запази разнообразието на поколението, като по този начин намаля шансовете на алгоритъма да популацията да се изроди. Мутацията може да бъде извършена по много начини. Най-простото е случаен смяна на 2 елемента (или

промяна на булева стойност при булевите хромозоми), според начина на представянето на състоянията.

### **Ход на алгоритъма.**

**Инициализация.** Първоначално много различни решения се генерират случајно и сформират популацията. Размерът на популацията зависи от естеството на проблема, но обикновено съдъжа няколко стотин до няколко хиляди възможни решения. Традиционно популацията се генерира случајно, позволявайки за да може да се засегне по-голяма част от пространството на решенията. В някои случаи, решенията могат да бъдат концентрирани в никаква зона, за която предполагаме, че е най-вероятно да съдържа оптимални решения.

**Селекция.** По време на всяко поколение, част от популацията се избира да се „размножи“ и да образува следващото поколение. Индивидуални решения се избират, чрез функцията за оценка на приспособимостта, като се вземат тези с най-добрите показатели. Други методи пък, измерват случаен набор от индивиди от популацията, ако процесът по избиране на най-пригодните индивиди може да бъде много времетнемащ.

**Генетични операции.** Следващата стъпка е да се генерира новото поколение от решения на базата на селектиранияте индивиди. Тук идват кръстосването (рекомбинация) и/или мутацията. При кръстосването вземаме от подбранныте индивиди, като по този начин очакваме, че децата им ще наследяват и надминават техните добри качества и ще имат по-висока „приспособимост“. Въпреки, че ползването на 2ма родители е по-биологично „коректно“, според някои, ползването на повече родители в много случаи води до по-добри резултати.

Във всяко следващо поколение цялостната приспособимост на популацията трябва да се покачва, тъй като избираме само най-добрите индивиди за кръстосването.

Могат на случаен принцип да се прилагат мутации, за да се поддържа разнообразието в популацията. Въпреки, че мутацията може да причини „загуба“ на добри индивиди, тя може да спомогне за избягването на защътливие при уединяване на популацията.

За да запазим разнообразието в популацията, можем да прилагаме мутация.

**Приключване на процеса.** Процесът на генерация се повтаря до достигане на някои от терминалните критерии. Най-честите примери за такива са:

1. Определяне на минимален критерий и намиране на решение, което го удоволства
2. Фиксиран брой поколения
3. Предварително фиксиран бюджет (изчислително врем / пари)
4. Приспособимостта на най-добрите индивиди достига плато и последващите итерации не дават по-добри резултати.
5. Ръчна инспекция.
6. Комбинация от горните

## **6.4 Речник**

Local search	Локално търсене
Greedy	Лакомо
Randomized	Рандомизиран / случаен
Hill-climbing	Катерене по хълм
Ridge	Хребет
Simulated annealing	Симулирано каляване
Crossover	Кръстосване

## **6.5 Ресурси**

„Local Search Algorithms for Combinatorial Problems”, Wolfgang Bibel

„Local Search Genetic Algorithm for Optimal Design of Reliable Networks”, Berna Dengiz and Fulya Altiparmak

“Simulated annealing”, Franco Busetti

“Simulated Annealing”, Dimitris Bertsimas and John Tsitsiklis

[http://en.wikipedia.org/wiki/Genetic\\_algorithm](http://en.wikipedia.org/wiki/Genetic_algorithm)

[http://en.wikipedia.org/wiki/Local\\_search\\_\(optimization\)](http://en.wikipedia.org/wiki/Local_search_(optimization))

[http://en.wikipedia.org/wiki/Hill-climbing\\_optimization](http://en.wikipedia.org/wiki/Hill-climbing_optimization)

<http://www.obitko.com/tutorials/genetic-algorithms/index.php>

<http://www2.econ.iastate.edu/tesfatsi/holland.gaintro.htm>

[http://www.scholarpedia.org/article/Genetic\\_algorithms](http://www.scholarpedia.org/article/Genetic_algorithms)

## 7 Удовлетворяване на ограниченията

### **Съдържание**

[Формулировка](#)

[Пример](#)

Оцветяване на карта

[Граф на ограниченията](#)

[Инкрементална формулировка](#)

[Типове променливи](#)

Независими променливи с крайни домейни

Независими променливи с безкрайни домейни

Променливи с непрекъснати домейни

[Типове ограничения](#)

Унарни ограничения

Бинарни ограничения

Ограничения от по-висок ред

Предпочитания

[Търсене с възврат](#)

Ред на променливите и стойностите

Разпространяване на информацията чрез ограничения

Интелигентно търсене с възврат

[Локално търсещи алгоритми](#)

[Структура на проблема](#)

[Литература](#)

### **7.1 Речник**

constraint satisfaction problem(CSP)	задача за удовлетворяване на ограниченията(ЗУО)
assignment	присвояване, свързване
consistent assignment	съвместимо свързване
legal assignment	легално свързване
objective function	целева функция
constraint graph	граф на ограниченията

successor function	функция, определяща наследниците
commutativity	комутативност
arc consistency	съвместимост на дъгите
backtracking search	търсене с възврат, търсене с връщане
branching factor	фактор на разклонение
constraint language	език на ограниченията
domain	домейн, област от допустими стойности, деф.област

## 7.2 Формулировка

Задачата за удовлетворяване на ограниченията се дефинира чрез: - множество от **променливи**  $X_1, X_2, \dots, X_n$  - множество от **ограничения**  $C_1, C_2, \dots, C_m$ . Всяка променлива  $X_i$  има съответна област от допустими стойности  $D_i$  ( $D_i \neq \emptyset$ ). Всяко ограничение  $C_i$  включва някакво подмножество от променливите и определя позволените комбинации от стойности за тях.

Състоянията представляват множество от свързвания със стойности на променливите,  $\{ X_i = v_i ; X_j = v_j ; \dots \}$ . Свързване, което не нарушава ограниченията, се нарича съвместимо или легално присвояване. Пълно свързване е онова, в което участват всички променливи. Решение на задачата за удовлетворяване на ограниченията е пълно свързване, което удовлетворява всички ограничения. Някои ЗУО изискват решение, което максимизира дадена целева функция.

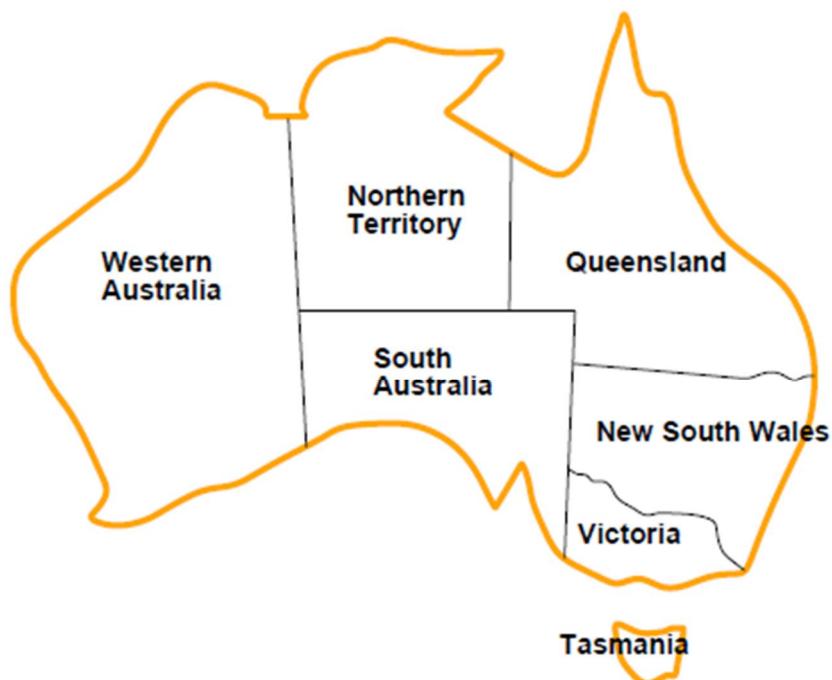
### 7.2.1 Пример: Оцветяване на карта

Нека е дадена картата на Австралия със съответните територии. Задачата е да се оцвети всеки регион с червено, зелено или синьо, така че съседни региони не са с еднакъв цвят.

Нека формулираме поставения проблем в термините на CSP:

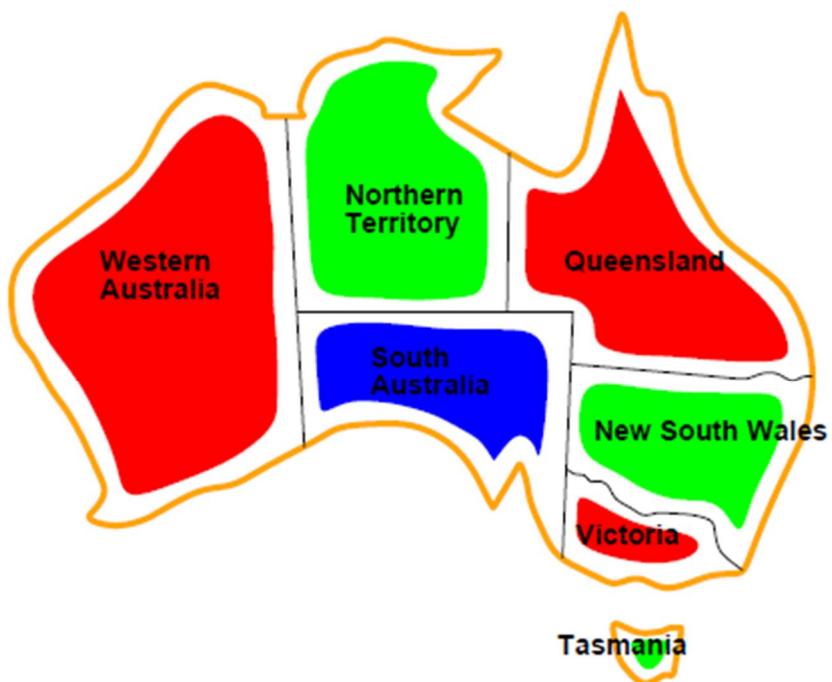
Променливите са съответните региони: WA, NT, Q, NSW, V, SA, T.

Области от допустими стойности за всяка променлива: {червено, зелено, синьо}.



източник <http://inst.eecs.berkeley.edu/>

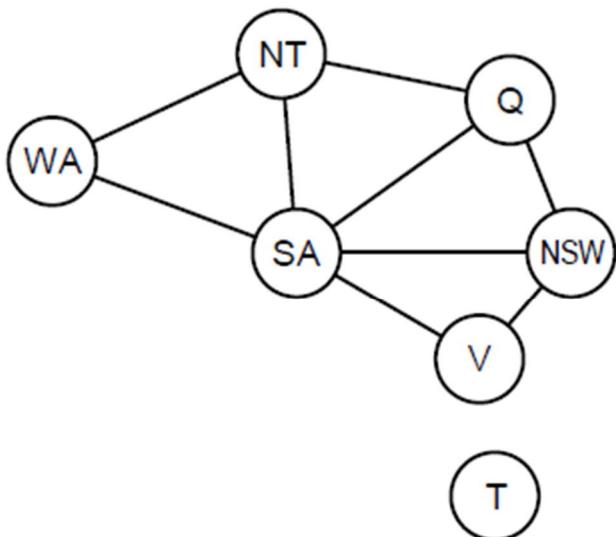
Ограниченията изискват съседни региони да имат различни цветове, например позволените комбинации за WA и NT са двойките {(червено, зелено); (червено, синьо); (зелено, червено); (зелено, синьо); (синьо, червено); (синьо, зелено)}. Ограниченията могат да бъдат представени и по-ясно като например  $WA \neq NT$ . Възможно решение е например  $\{WA = \text{червено}, NT = \text{зелено}, Q = \text{червено}, NSW = \text{зелено}, V = \text{червено}, SA = \text{синьо}, T = \text{червено}\}$ .



източник <http://inst.eecs.berkeley.edu/>

### 7.2.2 Граф на ограниченията

ЗУО могат да се представят посредством **граф на ограниченията**. На фигурата по-долу е показан графът на ограниченията за разгледания пример. Възлите на графа отговарят на променливите на задачата, а дългите - на ограниченията. Задачите за удовлетворяване на ограниченията с общо предназначение използват графичната структура, за да ускорят търсенето. В разгледания пример Тасмания се оказва независим подпроблем.



източник <http://aima.cs.berkeley.edu/newchap05.pdf>

### 7.2.3 Инкрементална формулировка

Може да забележим, че при задачата за удовлетворяване на ограничения може да се приложи инкрементална формулировка като при стандартна задача за търсене, както следва:

Начално състояние: на променливите не са присвоени стойности, {}

Функция, определяща наследниците: на променлива без стойност ѝ се присвоява такава, но не може да бъде в конфликт с предишни свързани променливи

Цел: текущото присвояване да е пълно

Цена на пътя: константна цена (например 1) за всяка стъпка

Всяко решение трябва да бъде пълно свързване и следователно се намира на дълбочина  $n$ , ако има  $n$  променливи. Още повече, дървото на търсене се разширява само до дълбочина  $n$ . Поради тази причина, алгоритмите за търсене в дълбочина са популярни при ЗУО. Също така пътя до целта е неревантент. Тогава може винаги да се използва формулировка, при която всяко състояние има стойност, която може или не да удовлетворява ограниченията.

### 7.2.4 Типове променливи

**Независими променливи с крайни домейни** Това е най-простият вид ЗУО. Оцветяването на карта е от този тип. Задачата за 8те царици също може да бъде разглеждана като такъв вид ЗУО, където променливите  $Q_1, \dots, Q_8$  са позициите на всяка царица в колоните 1,, 8 и всяка

променлива има домейн  $\{1, 2, 3, 4, 5, 6, 7, 8\}$ . Ако максималния размер на дефиниционната област на някоя променлива е  $d$ , тогава броят на възможните пълни свързвания е  $O(d^n)$  – тоест експоненциален по броя на променливите.

Разгледаният вид ЗУО включват и **булеви ЗУО**, чиито променливи могат да приемат стойности true или false. Булевите ЗУО включват като специални случаи някои NP-пълни проблеми, като например 3SAT.

**Независими променливи с неограничени домейни** Независимите променливи могат да имат и неограничени дефиниционни области - например, множеството на целите числа или на стринговете. Например, при съставяне график на работа, началната дата е променлива и възможните ѝ стойности са числа (дните след текущата дата). При тези ЗУО не е възможно да се опишат ограниченията посредством изброяване на всички позволени комбинации от стойности, вместо това трябва да се използва *език на ограниченията*. Например, ако *Rабота1*, отнемаша 5 дни, трябва да предшества *Rабота3*, то ни трябва език на ограниченията с математически неравенства като  $Rабота1 + 5 \leq Rабота3$ . Може да сведем тези задачи до ЗУО с ограничени домейни посредством налагане на допълнителни ограничения на всички променливи. Например при графика за работата може да поставим горна граница, равна на всички задачи, които трябва да се насрочат.

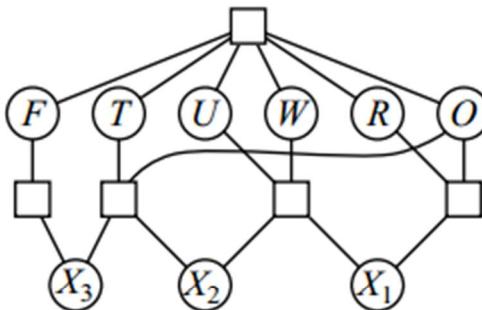
**Променливи с непрекъснати домейни** ЗУО с непрекъснати домейни са често срещани в реалността и са широко изучавани в областта на изследване на операциите. Например, планирането на експерименти с космическия телескоп Хъбъл изиска много прецизно време на наблюденията; началото и краят на всяко наблюдение и маневрите са променливи, които са непрекъснато оценявани и се подчиняват на разнообразие от астрономически ограничения и ограничения в мощността.

Най-добре известната категория такива ЗУО е тази на проблемите в линейното програмиране, където ограниченията трябва да са линейни неравенства и формиращи изпъкналост. Задачите в линейното програмиране могат да бъдат решени в полиномиално време на броя на променливите.

### 7.2.5 Типове ограничения

**Унарни ограничения** – ограничават стойността на една единствена променлива. За примера с картата,  $SA \neq green$ . **Бинарни ограничения** – свързват 2 променливи. Например  $SA \neq NSW$ . Бинарна ЗУО е ЗУО, която се състои само от бинарни ограничения. Може да се представи чрез граф на ограниченията. **Ограниченията от по-висок ред** – включват 3 и повече променливи. Например, криптоаритметични пъзели (фиг. 4a). Обикновено всяка буква в пъзела представя различна цифра. Ограниченията от по-висок ред могат да бъдат представени чрез хиперграф на ограниченията (фиг. 4б).

$$\begin{array}{r}
 T \ W \ O \\
 + T \ W \ O \\
 \hline
 F \ O \ U \ R
 \end{array}$$



(a)

(b)

**Фиг.4** (а) Криптоаритметичен пъзел. Всяка буква отговаря за различна цифра; целта е да се намери такова заместване на буквите с цифри, че изразът да бъде математически коректен; не е позволено да се започва с нули. (б) Хиперграф на ограниченията за криптоаритметичния пъзел; всяко ограничение е квадратче, свързано с променливите, които ограничава.

източник <http://aima.cs.berkeley.edu/newchap05.pdf>

**Предпочитания (слаби ограничения)** – определят критерии за избор между няколко решения (дефинират цената на някои свързвания на променливи)

### 7.3 Търсене с възврат

Да предположим, че приложим търсене в ширина към общата формулировка на проблема в предишната точка. Забелязваме следното: факторът на разклонение на най-високото ниво е  $nd$ , защото всяка от  $d$ -те стойности може да бъде присвоена на всяка от  $n$ -те променливи. На следващото ниво този фактор е  $(n-1)d$  и така за  $n$  нива. Генерирахме дърво с  $n!d^n$  листа, макар и да има само  $d^n$  възможни пълни присвоявания!

Нашата привидно логична формулировка на задачата е игнорирана много важно свойство, характерно за всички ЗУО: **комутативността**. Една задача е комутативна, ако реда на прилагане на даден набор от действия не оказва влияние върху резултата. Такъв е и случаите с ЗУО, защото, когато присвояваме стойности на променливите, достигаме същото частично присвояване, независимо от реда. Следователно, всички алгоритми за търсене за ЗУО генерират наследници, вземайки под предвид възможните присвоявания само за една променлива на всеки възел в дървото. Например, в корена на дървото на търсенето за оцветяването на картата на Австралия, може да избираме между  $SA = \text{червено}$ ,  $SA = \text{зелено}$ ,  $SA = \text{синьо}$ , но не може да избираме между  $SA = \text{червено}$  и  $WA = \text{синьо}$ . С това ограничение, броят на листата е  $d^n$ .

Понятието **търсене с възврат** се използва за търсене в дълбочина, което избира стойности за една променлива в даден момент и се връща, когато за дадена променлива няма правилни стойности за присвояване (фиг. 5). Забележете, че разширява текущото свързване, за да генерира наследник, а не да го копира. Тъй като представянето на ЗУО е стандартизирано, няма нужда да имаме начално състояние със специфичен домейн, функция, определяща наследниците или проверка за цел.

Обикновеното търсене с връщане е неинформиран алгоритъм, така че не очакваме да бъде много ефективен при големи задачи.

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
    return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns a solution, or failure
    if assignment is complete then return assignment
    var  $\leftarrow$  SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment according to CONSTRAINTS[csp] then
            add {var = value} to assignment
            result  $\leftarrow$  RECURSIVE-BACKTRACKING(assignment, csp)
            if result  $\neq$  failure then return result
            remove {var = value} from assignment
    return failure
```

Фиг. 5 Търсене с възврат - алгоритъм

Използването на общи методи (какъвто е текущо разглежданият) поставя редица въпроси, свързани с ефективността на търсения: • Коя променлива трябва да бъде свързана най-напред? • В какъв ред трябва да бъдат пробвани различните допустими стойности на избраната променлива? • Може ли да се прецени предварително дали дадено свързване ще доведе до неуспех?

Последващите точки ще отговорят на тези въпроси.

### 7.3.1 Ред на променливите и стойностите

Алгоритъмът за търсене с възврат съдържа следния ред

*var*  $\leftarrow$  SELECT-UNASSIGNED-VARIABLE(VARIABLES[*csp*], *assignment*, *csp*).

По подразбиране, *SELECT-UNASSIGNED-VARIABLE* избира следващата несвързана променлива в реда, зададен от списъка *VARIABLES[csp]*. Този статичен ред на променливите рядко резултира в най-ефективното търсене. Например, след присвояванията за *WA* = *чевено* и *NT* = *зелено*, има само една възможност за *SA*, така че е смислено да присвоим стойност (*синьо*) на *SA*, отколкото да присвоим стойност на *Q*. Тази идея, най-напред да се избере за свързване онази променлива, която има най-малък брой допустими стойности, се нарича *minimum remaining values (MRV)* евристика, или **“избор на най-ограничената променлива”**. Ако има променлива X с нито една възможна стойност, тази евристика ще избере X и съответно ще имаме неуспех. Изгълнението е от 3 до 3,000 пъти по-добро от това на обикновеното търсене с възврат в зависимост от задачата.

Да се върнем на примера с картата - MRV евристиката не помага особено в избирането на първия регион, който да бъде оцветен, защото първоначално за всеки регион има по 3 различни цвята. В този случай може да се позовем на *degree* евристиката, или **“избор на най-ограничаващата променлива”**. Най-напред се избира за свързване онази променлива, която ще наложи най-много ограничения върху оставащите несвързани променливи. *SA* е променливата с най-високата степен - 5, другите имат степен 2 или 3, а *T* - 0. Веднъж след като

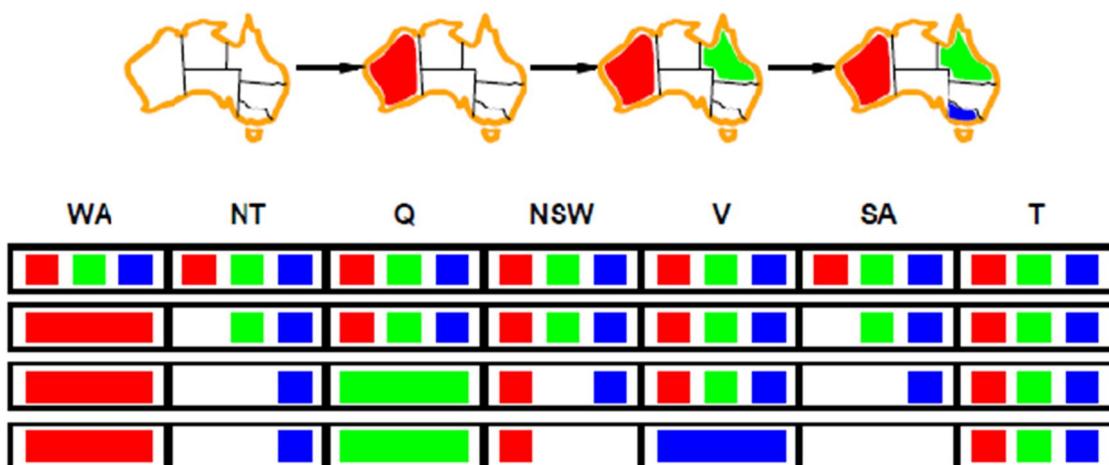
е избрана SA, степенната евристика решава проблема без грешни стъпки - може да изберем всеки съвместим цвят на всяка стъпка и да стигнем до решение без връщане.

След като е избрана променлива, алгоритъмът трябва да реши в какъв ред да изследва съответните стойности. Тук много ефективна може да бъде евристиката за **“избор на най-малко ограничаващата променлива”**. Най-напред се избира за свързване онази променлива, която ще наложи най-малко ограничения върху оставащите несвързани променливи. Ако искаме обаче да открием всички решения на дадена задача, а не само първото, то тогава редът няма значение, тъй като ще трябва да пробваме всички варианти. Същото се отнася и при липса на решение на проблема.

### **7.3.2 Разпространяване на информацията чрез ограничения**

Дотук нашият алгоритъм за търсене отчита ограниченията върху променлива само в момента, в който тя е избрана от *SELECT-UNASSIGNED-VARIABLE*. Но преглеждайки някои от ограниченията по-рано в търсенето, или дори преди то да започне, може драстично да намалим пространството на търсенията.

**Forward checking** Един начин да се възползваме по-добре от ограниченията по време на търсенето е forward checking. Когато променлива X е свързана, forward checking процесът преглежда всяка неприсвоена променлива Y, която е свързана посредством ограничение с X, и изтрива от домейна на Y стойностите, които са несъвместими със стойността, избрана за X. Пример: оцветяване на карта фиг. 6.



Фиг. 6 Оцветяване на карта с forward checking - първо се свързва WA = червено; forward checking изтрива червено от домейните на съседните променливи NT и SA; след Q = зелено, зелено е изтрито от домейните на NT, SA и NSW; след V = синьо, синьо е изтрито от домейните на NSW и SA, оставяйки SA без възможни стойности.

източник <http://inst.eecs.berkeley.edu/>

### **Разпространяване на ограниченията**

Макар че forward checking прихваща много несъвместности, не прихваща всички от тях. За примера с картата, когато WA=червено и Q=зелено, NT и SA трябва да са сини, но те са съседни райони. Forward checking не прихваща това, тъй като не преглежда толкова напред. Разпространяването на ограничения е общо понятие за разпространяване на ограничения от

една променлива към други. Разпространяването на ограниченията изисква многократно засилване на ограниченията на локално равнище. В разглежданият пример, трябва да засилим WA и Q към NT и SA (ако бе направено от forward checking) и после към ограничението между NT и SA, за да прихванем несъвместимостта.

**Съвместимост на дъгите** Идеята за съвместимост на дъгите осигурява бърз метод за разпространение на ограниченията, който е значително по-силен от forward checking. Терминът дъга се отнася към насочената дъга в графа на ограниченията (например дъгата от SA до NSW). Имайки текущите домейни за SA и NSW, може да определим, че дъгата е съвместима, ако за всяка стойност  $x$  на SA има стойност  $y$  на NSW, която е съвместима с  $x$ . От този ред на фиг.6, домейните на SA и NSW са съответно {синьо} и {синьо, червено}. За  $SA = \text{синьо}$  има съвместимо свързване за  $NSW = \text{червено}$   $\Rightarrow$  дъгата от SA към NSW е съвместима, но пък за дъгата  $NSW \rightarrow SA$  не е.

Може да приложим съвместимост на дъгите и към  $SA \rightarrow NT$  на същия етап от търсенето. И двете променливи имат домейн {синьо} и резултатът е, че синьото трябва да бъде изтрито от дефиниционната област на SA, оставяйки я празна. Така, прилагайки този алгоритъм, прихващаме несъвместимостите, които не са прихванати от forward checking.

Проверката за съвместимост на дъгите може да бъде приложена преди началото на търсенето или след всяко присвояване по време на търсенето (Maintaining Arc Consistency algorithm). Във всеки случай процесът трябва да се прилага многократно, докато не останат несъвместимости. Това се налага, защото при изтриване на стойност от домейн на някоя променлива с цел премахване на несъвместимостта, ново противоречие може да възникне към дъгата, сочеща към тази променлива. Пълният алгоритъм, AC-3<sup>1</sup>, използва опашка, за да съхранява дъгите, които трябва да бъдат проверени за неконсистентност (фиг. 7).

```

function AC-3(csp) returns the CSP, possibly with reduced domains
  inputs: csp, a binary CSP with variables  $\{X_1, X_2, \dots, X_n\}$ 
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
     $(X_i, X_j) \leftarrow \text{REMOVE-FIRST}(\text{queue})$ 
    if REMOVE-INCONSISTENT-VALUES( $X_i, X_j$ ) then
      for each  $X_k$  in NEIGHBORS[ $X_i$ ] do
        add  $(X_k, X_i)$  to queue

function REMOVE-INCONSISTENT-VALUES( $X_i, X_j$ ) returns true iff we remove a value
  removed  $\leftarrow \text{false}$ 
  for each  $x$  in DOMAIN[ $X_i$ ] do
    if no value  $y$  in DOMAIN[ $X_j$ ] allows  $(x, y)$  to satisfy the constraint between  $X_i$  and  $X_j$ 
      then delete  $x$  from DOMAIN[ $X_i$ ]; removed  $\leftarrow \text{true}$ 
  return removed

```

**Фиг. 7** Алгоритъм съвместимост на дъгите, AC-3. След прилагането му, или всяка дъга е съвместима, или някоя променлива има празен домейн, индикатор, че ЗУО не може да бъде решена.

източник <http://aima.cs.berkeley.edu/newchap05.pdf>

<sup>1</sup> Името AC-3 е използвано от откривателя на този алгоритъм, Mackworth, 1977, което въобще означава 3-та версия на разработка.

Всяка дъга  $(X_i; X_j)$  бива изтрита и проверена; ако някоя стойност трябва да бъде изтрита от ДО на  $X_i$ , тогава всяка дъга  $(X_k; X_i)$ , сочеща към  $X_i$  трябва да бъде отново добавена в опашката за проверка. Сложността на проверката за съвместимост на дъгите може да бъде анализирана по следния начин: бинарна ЗУО има най-много  $O(n^2)$  дъги; всяка дъга  $(X_k; X_i)$  може да бъде включена само  $d$  пъти, тъй като  $X_i$  има най-много  $d$  стойности за изтриване; проверката за съвместимост на дъга може да бъде извършена за време  $O(d^2)$ ; така в най-лошия случай ще имаме време  $O(n^2d^3)$ . Макар това да е значително по-скъпо от forward checking-a, понякога допълнителната цена си заслужава.

Проверката за съвместимост на дъгите не открива всяко несъответствие. Например, частичното свързване  $\{WA = \text{червено}, NSW = \text{червено}\}$  е несъвместимо, но AC-3 няма да го прихване.

### 7.3.3 Интелигентно търсене с възврат

Алгоритъмът за търсене с възврат има много проста логика какво да прави при провал на част от търсенето: връща се към предпешващата променлива и пробва друга стойност за нея. Това се нарича хронологично връщане, защото най-скорошния точка е посетена отново. Какво се случва обаче, ако приложим същата тази логика при фиксиран ред на променливите, а именно Q, NSW, V, T, SA, WA, NT. Да предположим, че сме генерирали частичното свързване  $\{Q = \text{червено}; NSW = \text{зелено}; V = \text{синьо}; T = \text{червено}\}$ . Когато пробваме следващата променлива SA, виждаме че всяка стойност наруши ограничение. Връщаме се назад, към T и пробваме друг цвят за Тасмания! Това очевидно е излишно, тъй като няма да има никакъв ефект.

По-интелигентен подход е да се върнем към някоя променлива, която е причинила неуспеха. Такова множество от променливи наричаме конфликтно. В примера, конфликтното множество е  $\{Q, NSW, V\}$ . По-общо, конфликтното множество за променлива X е множеството от предишни променливи, които са свързани посредством ограничение към X. Методът **backjumping** се връща към най-скорошната стойност от конфликтното множество. В примера, ще се върне към V и ще прескочи T. Наблюдалният читател може да е забелязал, че необходимостта от този метод се появява в момента, когато всяка стойност в ДО е в конфликт с текущите присвоения, но forward checking прихваща това и не позволява на търсенето дори да достигне този възел. И така, обикновеното “прескачане” става излишно при forward checking или при търсене, което използва по-силна проверка за съвместимост, каквато е MAC.

Въпреки това заключение, идеята, която стои зад този метод е добра: връщане назад, основано на причините за неуспеха. Да вземем отново предвид частичното свързване  $\{WA = \text{червено}, NSW = \text{червено}\}$ , което е несъвместимо. Нека опитаме  $T = \text{червено}$  и после да свържем NT, Q, V, SA. Знаем, че никое присвояване няма да сработи за тях, затова изчерпваме стойностите, които пробваме за NT. И сега въпросът е къде да се върнем? Прескачането не работи, защото NT има стойности, съвместими с предпешващите свързани променливи - NT няма пълно конфликтно множество от предпешващи променливи, които да водят до неуспеха. Но пък знаем, че 4те променливи NT, Q, V, SA, взети заедно, водят до неуспех, заради предпешващите ги. Това води до по-дълбок извод за конфликтно множество за променлива като NT: това е онова конфликтно множество, което заедно с NT и с която и да е

следваща променлива, довежда до липса на съвместимо решение. В разглеждания случай това е {WA, NSW}, така че алгоритъмът трябва да се върне към NSW и да прескочи Tasmania. Backjumping алгоритъм, който използва така дефинирано конфликтно множество, се нарича **conflict-directed backjumping**. Нека дефинираме как трябва да се конструира конфликтното множество: Нека  $X_j$  е текущата променлива и  $\text{conf}(X_j)$  да е нейното конфликтно множество. Ако всяка възможна стойност за  $X_j$  се провали, прескочи до най-скорошната променлива  $X_i$  от  $\text{conf}(X_j)$  и постанови

$$\text{conf}(X_i) \leftarrow \text{conf}(X_i) \cap \text{conf}(X_j) - \{X_i\}.$$

## 7.4 Локално търсещи алгоритми за ЗУО

Локално търсещите алгоритми се оказват много ефективни в решаването на ЗУО. Те използват следната формулировка: начално състояние - присвоява се стойност на всяка променлива, а функцията, определяща наследниците променя стойността на една променлива в даден момент. За пример да разгледаме задачата за 8те царици: началното състояние е случаино генерирана конфигурация на 8те царици в 8те колони, а функцията, определяща наследниците взема една царица и я мести на друго място в колоната. Друга възможност е да започнем с 8те царици, по една във всяка колона в пермутация на 8те реда и да генерираме наследник като разменяме редовете на 2 царици.

При избиране на нова стойност за променлива, най-видната евристика е изберем стойност, която е с минимален брой конфликти с другите стойности на променливите - т.нр. **min-conflicts** евристика. Min-conflicts алгоритъмът присвоява произволни стойности към на всички променливи от задачата за удовлетворяване на ограниченията. Следващата стъпка е да избере произволна променлива, чиято стойност обаче е в конфликт с някое ограничение. После присвоява на тази променлива стойността с минимум конфликти като: ако има повече от една такава стойност, избира една от тях на произволен принцип. Стартита се нова итерация със същите стъпки и така, докато се намери решение или се достигне максималния брой итерации (предварително зададен).

Този алгоритъм може да се разглежда като евристика, тъй като ЗУО може да се интерпретира като локално търсещ проблем. Изненадващо, при задачата с n-те царици, ако не броим първоначалната подредба на цариците, времето му за изпълнение е (грубо казано) независимо от размера на задачата. Може да реши дори и million-queens problem в средно 50 стъпки (след първоначалната инициализация). Min-conflicts алгоритъмът е показан на фиг.8, а на фиг. 9 може да видите решение на задачата за 8те царици в 2 стъпки с този алгоритъм.

Друго предимство на локално търсещите алгоритми е, че могат да се използват, дори когато задачата се променя. Това е особено важно при задачи, свързани с планиране. Например, едноседмичното разписование на авиокомпания може да включва хиляди полети и 10 пъти по толкова персонални задачи, но лошото време само на едно летище може да направи това разписание неприложимо. Съответно искаме да поправим плана с минимум корекции. Това може лесно да бъде направено посредством локално търсещ алгоритъм, започвайки от текущото разписание. Търсенето с възврат с ново множество от ограничения обикновено изисква повече време и може да намери решение с много промени спрямо текущото разписание.

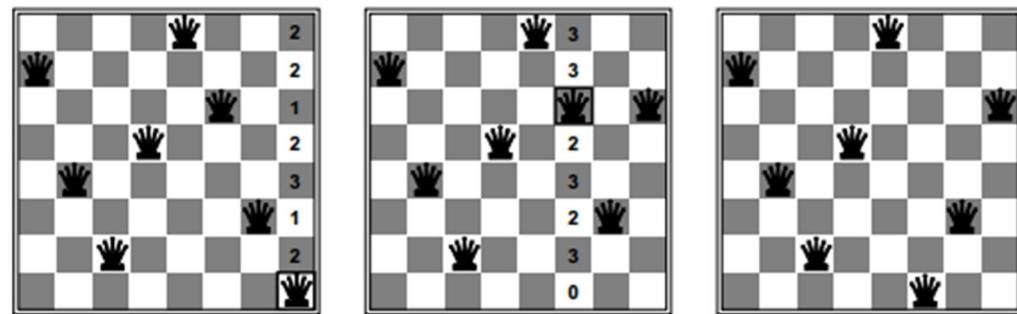
```

function MIN-CONFLICTS(csp, max_steps) returns a solution or failure
  inputs: csp, a constraint satisfaction problem
           max_steps, the number of steps allowed before giving up

  current  $\leftarrow$  an initial complete assignment for csp
  for i = 1 to max_steps do
    if current is a solution for csp then return current
    var  $\leftarrow$  a randomly chosen, conflicted variable from VARIABLES[csp]
    value  $\leftarrow$  the value v for var that minimizes CONFLICTS(var, v, current, csp)
    set var = value in current
  return failure

```

**Фиг.8** Min-Conflicts алгоритъм за решаване на ЗУО чрез локално търсене.  
източник <http://aima.cs.berkeley.edu/newchap05.pdf>



**Фиг. 9** Решение в 2 стъпки на задачата за 8те царици посредством min-conflicts алгоритъм.  
източник <http://aima.cs.berkeley.edu/newchap05.pdf>

## 7.5 Структура на проблема

В тази секция ще изследваме начини, при които структурата на проблема, представена чрез граф на ограниченията, може да се използва за бързо намиране на решения. Повечето от подходите тук са много основни и приложими и към други проблеми освен ЗУО. Все пак, единственият начин да се справим с реалните задачи, е да ги декомпозираме на много подзадачи. Ако се върнем към задачата за оцветяване на карта, си правим извода, че Тасмания не е свързана с останалата част от картата  $\Rightarrow$  става ясно, че оцветяването на Тасмания и на останалите региони, са независими подпроблеми. Независимостта може да се установи от свързаните компоненти в графа на ограниченията. Всяка компонента отговаря за един подпроблем - CSP<sub>i</sub>. Ако свързване S<sub>i</sub> е негово решение, тогава Ω<sub>i</sub> | S<sub>i</sub> е решение на Ω<sub>i</sub> | CSP<sub>i</sub>.

Зашто това е важно? Да предположим, че всяка CSP<sub>i</sub> има *c* променливи от общо *n* променливи, където *c* е константа. Тогава имаме *n/c* подзадачи, всяка от които ѝ отнема най-много *d<sup>c</sup>* за решаване. Следователно, общото време за работа е O(d<sup>c</sup>n/c), което е линейно по *n*, а без декомпозирането - O(d<sup>n</sup>), което е експоненциално по *n*. Независимите подпроблеми обаче са рядкост, в повечето случаи подзадачите са свързани.

Най-простият случай е, когато графът на ограниченията формира дърво: всеки 2 променливи са свързани с най-много 1 път. Ще покажем, че всяка ЗУО с дърводидна структура, може да бъде решена в линейно време по броя на променливите. Алгоритъмът е следният:

Избери която и да е променлива за корен на дървото, подреди променливите от корена до листата като всеки родител на връх в дървото го предшества в подредбата. Означи променливите с  $X_1; : : : ; X_n$ . Така всяка променлива, с изключение на корена, има точно 1 родител.

За  $j$  от  $n$  до 2, приложи съвместимост на дъгите към дъга  $(X_i; X_j)$ , където  $X_i$  е родител на  $X_j$ , изтривайки стойности от  $\Delta O$  на  $X_i$ , колкото е необходимо.

За  $j$  от 1 до  $n$ , свържи произволна стойност за  $X_j$ , която е съвместима със стойността на  $X_i$ , където  $X_i$  е родител на  $X_j$ .

Има две основни точки, които трябва да забележим. Първо, след стъпка 2 задачата за удовлетворяване на ограниченията е съвместима по отношение на дъгите в една посока, така че присвояването на стойности в стъпка 3 не изисква обратен ход (backtracking). Второ, прилагайки проверките за съвместимост на дъгите в обратен ред в стъпка 2, алгоритъмът осигурява, че никоя изтрита стойност не може да застраши съвместимостта на проверените вече дъги. Пълният алгоритъм се изпълнява за  $O(nd^2)$ . При наличието на ефективен алгоритъм за дървета, може да помислим дали по-общи графи на ограниченията могат да бъдат редуцирани до дървета. Съществуват два основни начина за това: един, базиран на изтриване на възли (*Cutset conditioning*) и другия, базиран на конструирането на дърводидна декомпозиция на графа на ограниченията на множество от свързани подзадачи.

## 7.6 Литература

Stuart Russell, Peter Norvig, [Artificial Intelligence: A Modern Approach \(2nd Edition\)](#), 2002

лекции Изкуствен интелект, спец. Информатика, <http://www.fmi.uni-sofia.bg/Members/marian>, 2012-13

<http://www.cs.toronto.edu/~bojajat/384n09/Lectures/Lecture-04-Backtracking-Search.pdf>

[http://en.wikipedia.org/wiki/Constraint\\_Satisfaction\\_Problems](http://en.wikipedia.org/wiki/Constraint_Satisfaction_Problems)

[http://en.wikipedia.org/wiki/Min-conflicts\\_algorithm](http://en.wikipedia.org/wiki/Min-conflicts_algorithm)

## **8 Избор на стратегия при игра за двама играчи - минимаксна процедура и алфа-бета отсичане**

### **Съдържание**

[Съдържание](#)

[Въведение](#)

[Минимаксна процедура](#)

[Примерна реализация на минимакс алгоритъма](#)

[\$\alpha\$ - \$\beta\$  отсичане](#)

[Примерна реализация на алфа-бета отсичане](#)

[Речник](#)

[Литература](#)

### **8.1 Увод**

Следващият основен тип задачи за търсене са задачите за намиране на печеливша стратегия в игра. От гледна точка на ИИ игрите се разделят на няколко типа в зависимост от това дали винаги настъпва край на играта, дали развитието ѝ зависи от случайността и дали играчите имат пълна информация за предишните и възможните бъдещи ходове на противника:

	Има ли край играта	Шанс
Пълна информация	Шах, Дама, Го, Отело	Табла, монополи
Непълна информация	„Stratego”, „Kriegspiel”	Бридж, покер, Скрабъл

Ще разгледаме само така наречените интелектуални игри с пълна информация с 2-ма играчи. Това са игри, за развитието на които не оказват влияние случайни фактори и във всеки един момент всеки от играчите разполага с информация за предишните и възможните бъдещи ходове на другия играч.

Както се вижда от таблицата по-горе шахът и дамата са такива игри. Докато всички игри с карти в дадена степен са въпрос на шанс как ще бъдат разпределени картите между играчите.

Основният проблем, който ИИ разглежда, е намирането на стратегия, която да доведе до победа, независимо от играта на противника. Затова за игрите, в които нямаме пълна информация за ходовете на противника или шансът оказва вклияние, би било практически невъзможно да се изгради стратегия, която винаги да е печеливша.

В настоящата глава ще разгледаме 2 алгоритъма за намиране на най-добър първи ход на текущия играч при текущото състояние на играта. Това е достатъчно за решаването на общата задача, тъй като тя може да бъде сведена до многократно намиране на най-добър първи ход.

Алгоритмите, които ще разгледаме, са минимакс процедура и  $\alpha$ - $\beta$  процедура. И двата алгоритъма намират стратегия при оптimalна игра на играчите. Оптimalна игра би означавало, че във всеки момент играчът прави най-добрния възможен ход. И двата алгоритъма работят върху дърво на състоянията ( $\Delta C$ ), изградено от възможните позиции в резултат на възможните ходове на играчите.

## 8.2 Минимаксна процедура

Минимаксната процедура се характеризира с това, че за да се изгради стратегия е необходимо първо да се построи цялото  $\Delta C$  и да се направи оценка на листата. От тази гледна точка за игри, в които  $\Delta C$  има много разклонения, минимаксната процедура не би била оптimalна и би изисквала много ресурси.

Минимаксната процедура завършва винаги, когато  $\Delta C$  е крайно.

Тя е оптimalна, когато играта на опонента също е оптimalна.

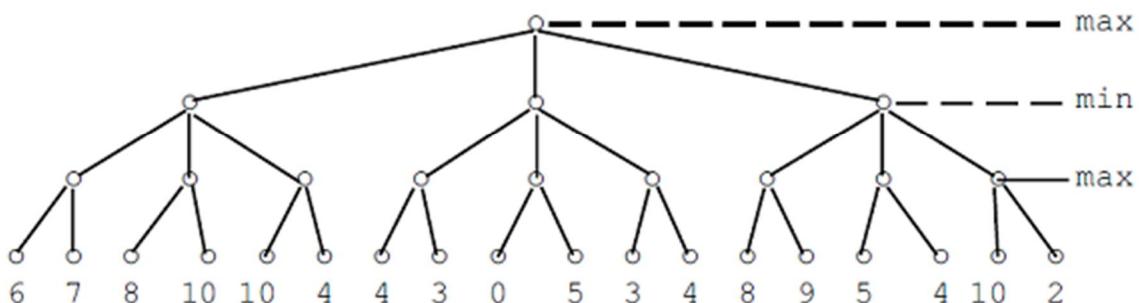
Времевата сложност на процедурата е  $O(b^m)$ , където  $b$  е коефициентът на разклоняване, а  $m$  е височината на дървото.

Необходима памет:  $O(bm)$ .

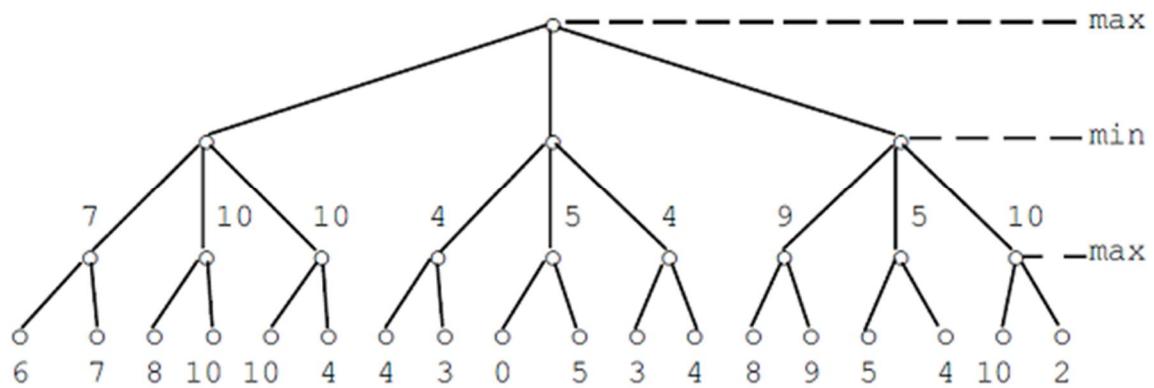
Приема се, че в рамките на играта двамата играчи имат противоположни цели. Единият се опитва да стигне до листо с максимална оценка, затова неговата стратегия е максимираща и той се нарича максимиращ играч. Целта на другия играч е да стигне до листо с минимална оценка и съответно неговата стратегия е минимираща, а той се нарича минимиращ играч (от там идва и името на процедурата). Минимаксната процедура оценява възлите от по-горните равнища на  $\Delta C$  и по-този начин дава възможност на първия играч да избере най-добрия си ход.

Ще дадем пример за това как минимаксната процедура оценява възлите на  $\Delta C$ .

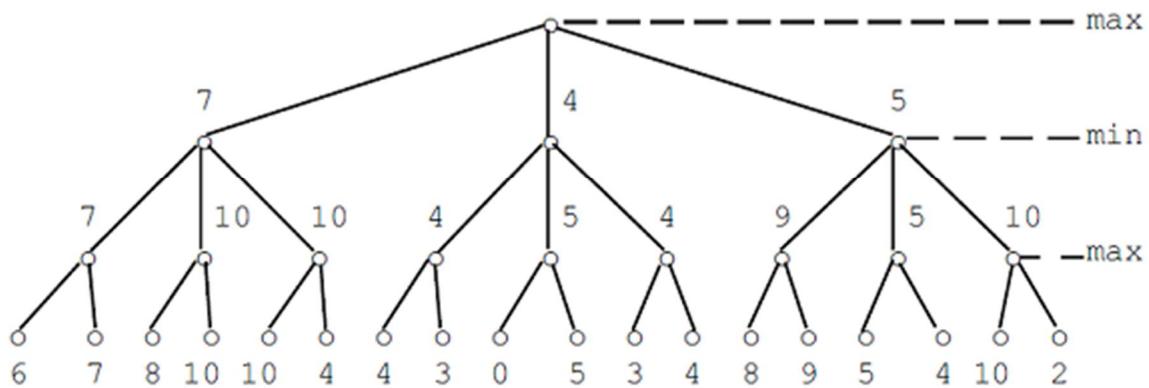
Нека разгледаме следното  $\Delta C$ , като на първа стъпка имаме само оценката на листата му. За определеност ще приемем че първи на ход е максимиращия играч.



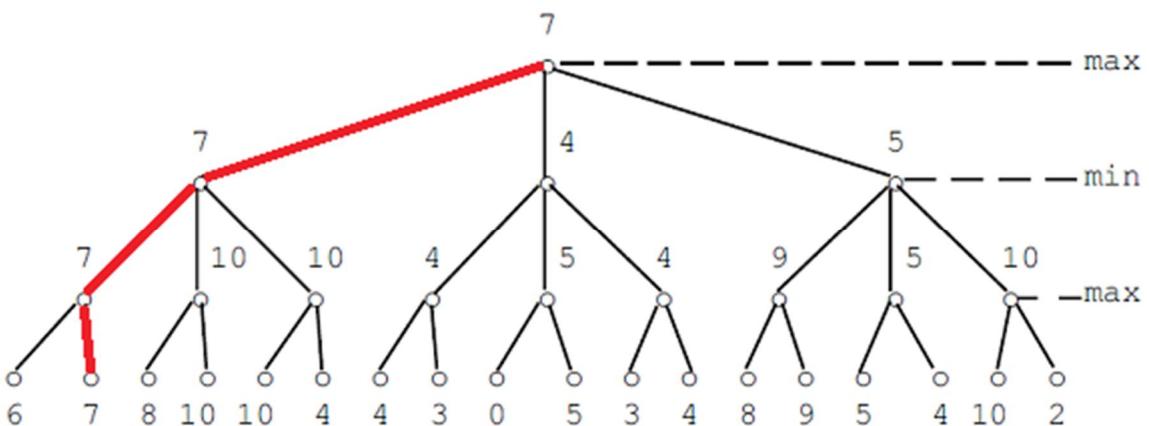
Отстрани е описано кой играч е на ход. Тъй като на предпоследното ниво максимиращият играч е на ход, то той ще търси листо с максимална оценка. Затова оценките на върховете на предпоследното ниво са следните:



На горното ниво на ход е минимизиращият играч. Той се стреми към следващо състояние, в което имаме минимална оценка. Затова оценката на възлите на дървото на това ниво ще е следната:



Следователно първият ход на максимизиращият играч става ясен. Той ще е в посока максимално текущо състояние и пълното дърво на състоянията става следното:



С червено е означена печелившата стратегия за първия играч, ако това е максимизиращият играч.

### Примерна реализация на минимакс алгоритъма

```
function Minimax-Decision(state) returns an action
```

```

inputs: state, current state in game
return the a in Actions(state) maximizing Min-Value(Result(a, state))
function Max-Value(state) returns a utility value
    if Terminal-Test(state) then return Utility(state)
    v ← -∞
    for a, s in Successors(state) do v Max(v, Min-Value(s))
    return v
function Min-Value(state) returns a utility value
    if Terminal-Test(state) then return Utility(state)
    v ← ∞
    for a, s in Successors(state) do v Min(v, Max-Value(s))
    return v

```

### 8.3 α-β отсичане

За разлика от минимакс алгоритъма,  $\alpha$ - $\beta$  процедурата няма нужда да построява цялото дърво на състоянията, за да намери стратегия. В момента в който се генерира ново състояние от ДС, то се оценява. Ако оценката му е по-добра от текущото състояние, дървото се отсича и се преминава към него. Какво значи оценката му да е по-добра зависи от това кой играч е на ход и ще бъде обяснено в последствие.

$\alpha$ - $\beta$  отсичането не променя финалния резултат, тоест сравнена с минимакс процедурата би дала същия резултат.

Тази процедура е по-бърза и по-ефективна спрямо минимакс процедурата.

Сложността ѝ по време е  $O(b^{m/2})$ .

Зашто процедурата се нарича  $\alpha$ - $\beta$ ?

$\alpha$  се нарича най-добрия връх за максимирация играч. Ако породената оценка на един връх е по-малка от  $\alpha$ , то максимиращия играч ще го избегне. Тоест за даден максимиращ връх  $\alpha$  е долна граница на породената от него оценка. Обратно -  $\beta$  е горната граница на породената оценка от минимиращ връх.

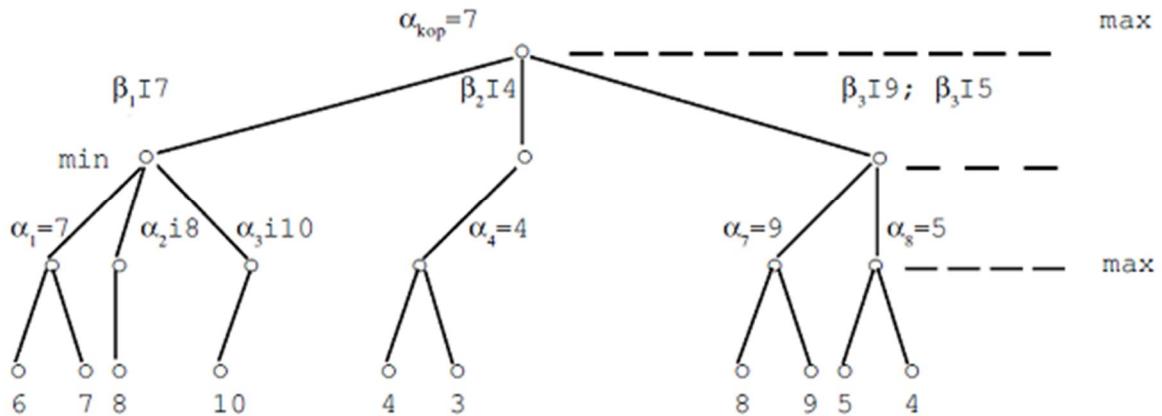
От тук следва че  $\alpha$  стойностите могат само да се увеличават в процеса на търсене на стратегия, а  $\beta$  стойностите могат само да намаляват.

По своя характер  $\alpha$ - $\beta$  отсичането прави обходждане на дървото в дълбочина.

Алгоритъмът извършва последователно  $\alpha$  и  $\beta$  отсичане от където идва и името му.  $\alpha$  отсичане се нарича отсичането на онези поддървета от ДС, които произлизат от минимален връх с  $\beta$  стойност по-малка или равна на  $\alpha$  стойността на съответния родител (който е максимален родител).  $\beta$  отсичане пък е противоположното – отсичане на онези поддървета от ДС, които произлизат от максимален връх с  $\alpha$  стойност по-голяма или равна на  $\beta$  стойността на съответния родител (който е минимален връх).

Тоест ако в процеса на генериране на възлите от ДС се окаже че от генерирането и оценяването на някои клонове за съответния играч няма да се получи по-добър резултат от вече получени, то тези клонове няма да се генерират изобщо.

Нека да разгледаме примера даден за минимакс процедурата, но вече прилагайки  $\alpha$ - $\beta$  отсичане.



Започваме обхождането на дървото винаги отдолу на горе и от най-левия му клон. На първата стъпка  $\alpha_1 = \max(6, 7) = 7$ . Тази оценка е точна, тъй като не може да се промени, защото е базирана върху оценките на всички подклонове (в случая това са само листата). Следователно можем да поставим неточна оценка на  $\beta_1$  на по-горния слой също 7, но трябва да проверим дали можем да намерим по добра оценка за  $\beta_1$ .  $\beta_1 = \min(\alpha_1, \alpha_2, \alpha_3)$ . Сега трябва да определим  $\alpha_2$  и  $\alpha_3$ , за да разберем дали можем да подобрим  $\beta$  оценката. За да определим  $\alpha_2$  посещаваме най-левия му клон, няма нужда да посещаваме и десния му, тъй като  $8 > \beta_1$  (бета отсичане). Аналогично намираме и  $\alpha_3 = 10$ . Следователно оценката за  $\beta_1$  няма да се промени и можем да преминем на горния слой. Вече сме на ниво корен на дървото.  $\alpha$  на корена е  $\max(\beta_1, \beta_2, \beta_3)$ . За да го оценим е необходимо по същата процедура да сметнем съответно  $\beta_2$  и  $\beta_3$ . В крайна сметка не успяваме да подобрим оценката за  $\alpha$ (корен) и по този начин вече сме намерили стратегия за максимизирана играч. Резултатът е същият като след минимакс процедурата, но докато в минимакс трябваше да обходим и разгледаме всичките 31 върха, тук беше достатъчно да генерираме само 27.

### Примерна реализация на алфа-бета отсичане

```

function Alpha-Beta-Decision(state) returns an action
    return the a in Actions(state) maximizing Min-Value(Result(a, state))
function Max-Value(state,  $\alpha$ ,  $\beta$ ) returns a utility value
    inputs: state, current state in game
         $\alpha$ , the value of the best alternative for max along the path to state
         $\beta$ , the value of the best alternative for min along the path to state
    if Terminal-Test(state) then return Utility(state)
    v  $\leftarrow -\infty$ 
    for a, s in Successors(state) do
        v  $\leftarrow \text{Max}(v, \text{Min-Value}(s, \alpha, \beta))$ 
        if v  $\geq \beta$  then return v
         $\alpha \leftarrow \text{Max}(\alpha, v)$ 
    return v
function Min-Value(state,  $\alpha$ ,  $\beta$ ) returns a utility value

```

same as Max-Value but with roles of  $\alpha$ ,  $\beta$  reversed

## 8.4 Речник

<i>space complexity</i>	сложност относно паметта
$\alpha$ - $\beta$ pruning	$\alpha$ - $\beta$ отсичане

## 8.5 Литература

Artificial intelligence a modern approach - Russell S., Norvig P.

Изкуствен интелект - М. Нинева, Д. Шипков, Интеграл 1995

[http://en.wikipedia.org/wiki/Minimax\\_algorithm](http://en.wikipedia.org/wiki/Minimax_algorithm)

[http://en.wikipedia.org/wiki/Alpha\\_beta\\_pruning](http://en.wikipedia.org/wiki/Alpha_beta_pruning)

## **9 Представяне и използване на знания - основни понятия и формализми.**

Представяне на знание? - „Науката как да представяме знание по начин, по който компютър може да се справи с него“. (Ръсел и Норвиг)

### **Съдържание**

[Съдържание](#)

[Какво е знание?](#)

[Разлики между данни и знания](#)

[Типове знания](#)

[Какво е представяне и използване на знания?](#)

[Разлики между бази от данни и бази от знания](#)

[Хипотеза за представянето на знания](#)

[Основни формализми за ПИЗ](#)

[Аспекти на формализмите за ПИЗ](#)

[Основни типове формализми за ПИЗ](#)

[Основни формализми за ПИЗ](#)

[Логическо представяне на знания](#)

[Какво е логика?](#)

[Съждителна логика](#)

[Логика от първи ред](#)

[Логики от по-висок ред](#)

[Други логики](#)

[Предимства и недостатъци на логическите системи](#)

[Семантични мрежи](#)

[Използване на знания, представени чрез семантични мрежи](#)

[Понятийни графи](#)

[Предимства и недостатъци на семантичните мрежи](#)

[Продукционни правила](#)

[Системи, основани на правила](#)

[Пример на система от продукционни правила](#)

[Предимства и недостатъци на продукционните правила](#)

[Фреймове](#)

[Представяне на знания чрез фреймове](#)

[Предимства и недостатъци на фреймовете](#)

[Характеристики на добрия формализъм за ПИЗ](#)

[Речник](#)

[Ресурси](#)

## **9.1 Какво е знание?**

Знание – мн. знания, ср. 1. Само мн. Съвкупност от факти от една област, които едно лице е усвоило.

В изкуствения интелект за разлика от лингвистична гледна точка под знание се разбира обобщена и формализирана информация за дадена предметна област. Тук следва да си зададем въпроса какво се разбира под информация, данни и знание и къде е разликата между тези понятия. Този въпрос води до редица философски разсъждения и анализи над езика, тъй като в системите с изкуствен интелект границата между тези понятия не е ясно изразена. Тук ще приемем, че данните и знанията са категории на понятието информация.

### **9.1.1 Разлики между данни и знания**

Данните са допълнителната информация за знанията; те носят информация за обектите; те са това, което наричаме факти в естествения език. Под факт ще разбираме общи известията в дадена предметна област истини или обстоятелства. Знанията от своя страна са общата и неизменна част от информацията; това са законите и взаимовръзките в самата информация. Например при пресмятане на дължината на конкретна окръжност данни за нас са конкретната дължина на радиуса и стойността на числото  $\pi$ . Знанието в този пример е формулата, по която пресмятаме дължината на окръжността:  $C=2 \times \pi \times r$ . С други думи знанието за нас е взаимовръзката между конкретните данни. Можем да обобщим, че върху данните се осъществяват действия, а знанията се използват за вземане на решения, изводи и съставяне на планове.

### **9.1.2 Типове знания**

Съществуват три основни типа знания в системите с изкуствен интелект:

1. Знания за обектите в предметната област;
2. Знания за връзките между обектите и знанията за тях;
3. Метазнания – знания за самите знания.

## **9.2 Какво е представяне и използване на знания?**

Представянето и използването на знания (ПИЗ) е област от изкуствения интелект, целта на която е знанието да се представи символно по такъв начин, който да улесни извеждането на заключения от знанията, с които разполагаме, използвайки по-скоро разсъждения отколкото

действия. Съществено за системите с изкуствен интелект е преминаването от работа с данни към работа със знания. Докато традиционните програмни системи работят с информация, организирана под формата на бази от данни (БД), то за системите с изкуствен интелект е характерно, че работят с бази от знания (БЗ).

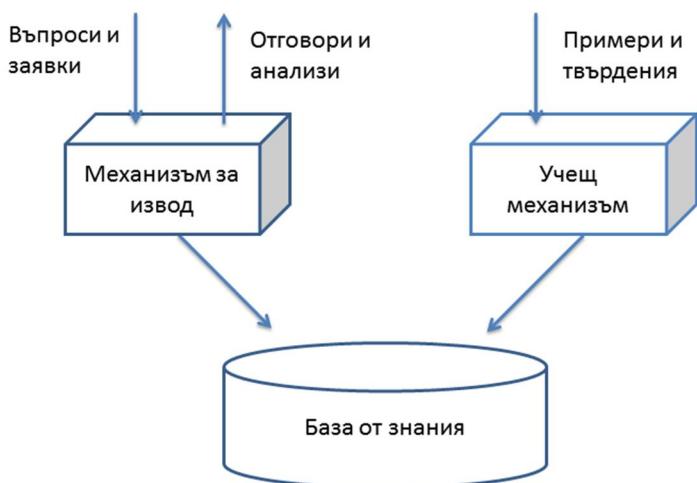
### **9.2.1 Разлики между бази от данни и бази от знания**

Докато базите от данни представляват съвкупност от данни и служат за извлечане само на такава информация, която е представена в явен вид в базата, то при базите от знания е възможно съставянето на нова информация, която не присъства в явен вид. Между данните в БД няма връзки, те изкуствено се създават в схема на данните, за да бъдат организирани в СУБД. За разлика от БД при БЗ съществуват връзки между знанията. Те могат да бъдат хоризонтални – отразяват ситуациянни отношения и причинно-следствени връзки, а също и вертикални – отразяват родово-видови връзки и наследяване на свойства. Базата от знания не е статична колекция от информация (каквато е базата от данни), а е динамичен ресурс (обикновено част от система с изкуствен интелект), който сам по себе си има възможността да учи и да извежда информация, препоръки и съвети. Накратко: БЗ дава отговори, а БД – просто списък от факти.

### **9.2.2 Хипотеза за представянето на знания**

Тази хипотеза е разработена през 1982г. от Брайън Смит. Същността на тази хипотеза гласи, че всяка система (естествена или изкуствена), която осъществява някакво интелигентно разсъждение, се състои от 2 структурно обособени части:

1. База от знания;
2. Машина за извод (интерпретатор) – обработва символите от БЗ с цел генериране на нови знания.



## **9.3 Основни формализми за ПИЗ**

### **9.3.1 Аспекти на формализмите за ПИЗ**

Всеки формализъм за представяне на знания може да бъде разглеждан в 2 основни аспекта: синтактичен и логически. Първият се отнася до начина, по който знанията се съхраняват в системата, а вторият – до начина, по който наличните знания могат да се използват за извеждане на допълнителни знания.

### **9.3.2 Основни типове формализми за ПИЗ**

Съществуват два основни типа формализми за ПИЗ - формализми от декларативен тип и формализми от процедурен тип. При първите съществен е начинът на представяне на знанията, докато при вторите съществен е начинът на използване на знанията. При декларативните формализми знанията се представят явно, а при процедурните, знанията са неявно представени чрез процедурите в програмната система. Изборът на типа формализъм зависи от конкретната задача.

### **9.3.3 Основни формализми за ПИЗ**

Съществуват 4 основни начина за ПИЗ:

1. Логическо представяне на знанията;
2. Семантични мрежи;
3. Продукционни правила;
4. Фреймове.

Ще разгледаме по-подробно всеки от тях.

## **9.4 Логическо представяне на знания**

Математическата логика е от декларативния тип формализми за ПИЗ.

### **9.4.1 Какво е логика?**

Основната цел на логиката е чрез средствата на формален език да се изрази знание за дадена предметна област. Логиката си служи с определени правила за извод. С тяхна помощ изразяваме и получаваме различни знания. Представянето на знания се осъществява чрез правилно построени формули в системата, а използването (извеждането) на знания се осъществява чрез правила и методи за извод. За логиката можем да си мислим като за език. В този ред на мисли има много начини да превеждаме от един език на друг. Каква част обаче от естествения ни език може да се преведе на езика на логиката?

Логиката като всеки друг език се характеризира със синтаксис и семантика. Под синтаксис разбираме начина, по който конструираме изреченията, а под семантика – начина, по който интерпретираме тези изречения. Нека разгледаме следния пример: „Всички лектори са високи 6 фута“. Това изречение само по себе си е напълно валидно от синтактична гледна точка. Освен това можем да разберем смисъла му, т.е. е и семантично валидно, но въпреки това изречението не носи вярна информация и го считаме за лъжа.

Нека се спрем по-подробно на някои видове логики.

### 9.4.2 Съждителна логика

#### Синтаксис

Основни понятия в съждителната логика са:

- Твърдения (Съждения) - бележат се с латински букви, например  $P$  и изразяват твърдения, например: „навън вали“. Съжденията могат да са верни (да приемат стойност Истина) или да са неверни (да приемат стойност Лъжа);
- Съждителни връзки: „ $\wedge$ “ (логическо и), „ $\vee$ “ (логическо или), „ $\neg$ “ (отрицание), „ $\rightarrow$ “ (импликация), „ $\leftrightarrow$ “ (еквивалентност);
- Скоби „(“, „)“.

#### Семантика

За да разберем смисъла на изречение, изказано чрез съждителна логика, трябва да сме наясно как различните съюзи влияят върху остойностяването на изречението съответно с истина или лъжа. Например:  $(P \wedge Q)$  е истина тогава и само тогава, когато и  $P$  и  $Q$  са истина. За извеждане на стойността на твърдението се използва следната таблица:

$P$	$Q$	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \rightarrow Q$	$P \leftrightarrow Q$
T	T	F	T	T	T	T
T	F	F	F	T	F	F
F	T	T	F	T	T	F
F	F	T	F	F	T	T

В нашия случай с T и F означаваме съответно истина и лъжа.

Представянето на знания чрез съждителна логика се осъществява чрез правилно построени формули (ППФ). Под ППФ разбираме последователности от твърдения, свързани с допустими съждителни връзки от съждителната логика.

### 9.4.3 Логика от първи ред

Логиката от първи ред е по-изразителна от съждителната.

#### Синтаксис

Синтаксисът на логиката от първи ред е разширение („надгражда“) този на съждителната логика, като въвежда още три понятия: термове, предикати и квантори. Понятието терм се определя по следния начин:

1. Константите са термове;
2. Променливите са термове;
3. Ако  $f$  е  $n$ -местен функционален символ, а  $t_1, t_2, \dots, t_n$  са термове, то  $f(t_1, t_2, \dots, t_n)$  също е терм;

4. Няма други термове освен построените чрез правилата 1-3.

Съществуват следните квантори: квантор за всеобщност ( $\forall$ ) и квантор за съществуване ( $\exists$ ). Чрез кванторите за всеобщност и съществуване можем да изразяваме, че дадено твърдение е вярно за всички обекти (твърдението е универсално) или че е вярно поне за 1 обект (твърдението е екзистенциално).

### **Семантика**

Семантиката при логиката от първи ред се изразява отново чрез правила за извод.

Нека отново се върнем към въпроса каква част от естествения ни език може да се преведе на езика на логиката? Нека разгледаме следния пример:

„Всеки понеделник и всяка сряда отивам при Джон за вечеря.“

На езика на предикатната логика от първи ред то би изглеждало по следния начин:

$\forall X ((\text{ден\_от\_седмицата}(X, \text{понеделник}) \vee \text{ден\_от\_седмицата}(X, \text{сряда})) \rightarrow (\text{отива\_при}(az, \text{Джон}) \wedge \text{яде}(az, \text{вечеря})))$

Важно е да отбележим, че се забелязва разлика при превода от „и“ в естествения език към логическо „или“.

### **9.4.4 Логики от по-висок ред**

Логиките от по-висок ред са още по-описателни и изразителни от тази от първи ред. В логиките от по-висок ред се въвежда възможността за квантифициране на функции, предикати, а също и на обекти.

### **9.4.5 Други логики**

Нека се спрем и на някои други видове логики:

#### **9.4.5.1 РАЗМИТА ЛОГИКА**

Размитата логика е формализъм за ПИЗ, който произлиза от теорията на размитите множества и чиято цел е да отразява неточни определения от типа на: „много“, „малко“, „повечето“. Размитата логика е вид многозначна логика, защото за разлика от класическата логика, при която боравим с 2 стойности – истина и лъжа (1 и 0), то тук боравим с вероятностни стойности, т.е. с променливи, остойностявани в интервала от 0 до 1. Под понятието многозначна логика разбираме такава, при която са допустими стойности, различни от традиционните - истина и лъжа.

#### **9.4.5.2 МОДАЛНА ЛОГИКА**

Модалната логика е формализъм за ПИЗ, който разширява съждителната и предикатната логика, като включва оператори, които изразяват модалности от типа на „необходимо“, „възможно“, „случайно“. Например ако вземем твърдението: „Джон е щастлив“, то може да го разширим с модалността: „обикновено“ по следния начин: „Джон обикновено е щастлив.“ Освен модалностите към този формализъм се добавя и още едно понятие – логическата операция строга импликация. Тя се означава с  $\Rightarrow$  и се дефинира по следния

начин:  $p \Rightarrow q$  е еквивалентно на  $\neg M(p \wedge \neg q)$ , където с  $Mp$  означаваме модалността:  $p$  е възможно. От това можем да забележим, че истинността на строгата импликация не зависи от стойността на  $p$ .

#### **9.4.5.3 ТЕМПОРАЛНА ЛОГИКА**

Темпоралната логика е формализъм за ПИЗ, който позволява представяне на знания чрез вземане предвид на времеви интервали. Например в темпоралната логика можем да изразяваме твърдения като: „Винаги съм гладен.“, „Ще съм гладен докато не се нахраня.“. Да вземем за пример твърдението: „Гладен съм.“ Въпреки, че значението на твърдението е константно във времето, истинността му може да варира във времето. Понякога твърдението е вярно, а понякога – не, но никога не приема стойностите истина и лъжа едновременно. Точно с това се характеризира темпоралната логика – истинността на твърдението се променя във времето.

#### **9.4.6 Предимства и недостатъци на логическите системи**

Едно от предимствата на логическите системи като формализъм за ПИЗ е, че преводът от естествен към формален език е сравнително лесен, заради ясната им семантика и изразителност. Има цели дялове в математиката, посветени на логиката, което я прави добър избор за представяне на знания в машинен вид. Освен това програмните езици сами по себе си са произлезли от логиката. В частност има програмни езици (като например Prolog), които използват предикатната логика, което значително улеснява представянето и използването на знания.

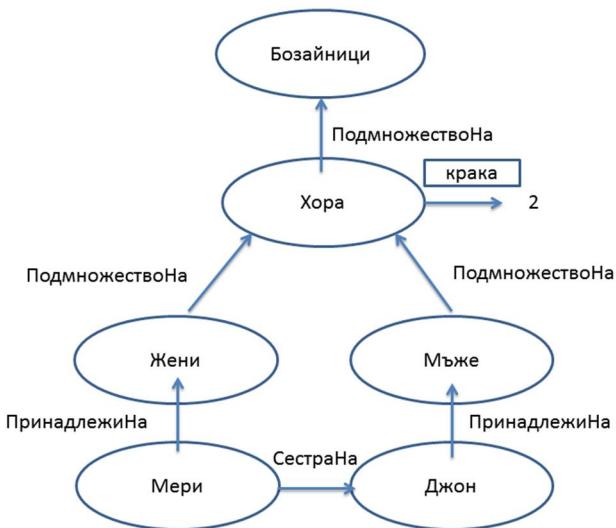
Недостатък на този формализъм е, че възникват проблеми при използване на неразрешими или полуразрешими логики. Такива са всички логики с крайна аксиоматика.

### **9.5 Семантични мрежи**

През 1909 Чарлз Пиръс предлага графична нотация на върхове и дъги, наречена екзистенциални графи, която той нарича „логиката на бъдещето“. Семантичните мрежи са от декларативния тип формализми за ПИЗ. Освен това те принадлежат към т.нр. структурни формализми, които се характеризират с това, че се основават на предположения за това, как знанията се запазват в човешкия мозък, а именно – под формата на отделни единици, свързани помежду си с връзки. Семантичните мрежи могат да се разглеждат като вид логика, като основно тяхно предимство пред логиките като формализъм за ПИЗ е, че нотацията им обикновено е доста по-подходяща за графично представяне.

Има много варианти на семантични мрежи, но в същината си всички те представлят знанията чрез обекти, категории обекти и връзки между обектите. Типичната графична нотация представя обектите и категориите в овали или правоъгълници и ги свързва с именувани дъги. Под обекти в случая разбираме понятията от предметната област, а дъгите са връзките (взаимоотношенията) между тези обекти.

Пример за семантична мрежа:



Важно свойство на семантичните мрежи е възможността за обратни връзки. Нека HasSister е обратният елемент на SisterOf, т.е.:  $\forall p,s \text{ HasSister}(p,s) \Leftrightarrow \text{SisterOf}(s,p)$ . Тогава ако имаме обект SisterOf, то можем да го свържем чрез такава връзка с обекта HasSister и така, ако търсим например сестрата на Джон, много по-ефективно и бързо можем да открием отговора, отколкото ако трябва да претърсим всички обекти SisterOf и за всеки от тях да проверяваме дали вторият аргумент е Джон.

### **9.5.1 Използване на знания, представени чрез семантични мрежи**

При този вид представяне на знания се извършват изводи за обектите, така че да получим неявно зададени техни свойства. Съществуват два основни механизма за извод върху семантичната мрежа: 1. разпространяваща се активност (търсене на сечение) и 2. наследяване.

1. Разпространяваща се активност (търсене на сечение) – състои се в това да се прави проверка дали в семантичните мрежи, описващи две понятия, се срещат общи такива. Този подход се използва най-често за установяване на сходство в значенията на две думи, проверявайки дали в семантичните мрежи на значенията на двете думи се съдържа едно и също понятие или накратко – търсим сечение на семантичните мрежи на двете понятия. Този подход се реализира по следния начин: Обхождаме семантичните мрежи на двете понятия, като на всяка стъпка достигаме нови възли и вдигаме техните флагове на активност. Търсенето приключва когато: или стигнем до възел, чийто флаг на активност е вдигнат (търсенето завърши с успех) или когато стигнем до изчерпване на възлите и на двете семантични мрежи (търсенето е завършило с неуспех).

2. Наследяване - състои се във възможността обектите да „наследяват“ свойства от категориите (класовете), към които принадлежат. Това се осъществява чрез причисляване на дадения клас към друг клас с помощта на връзки, например: E (ISA на англ.), Superclass, Instance, SubsetOf. Ако се върнем към примера за семантична мрежа по-горе можем да забележим, че ясно личи идеята за наследяване в графичното представяне на примера. Така например обектите Бозайници и Хора са свързани с връзката: Подмножество (SubsetOf), бихме могли да ги свържем и с ISA-връзка (e). Това означава, че по подразбиране Хората приемат дадени свойства на Бозайниците. По същия начин класът Хора има свойство: „Има

2 крака”. Знаейки това и вземайки предвид връзката Подмножество На от Мъже и Жени към Хора, лесно стигаме до извода, че Мъже и Жени притежават същото свойство. Подходът на наследяване се използва по следния начин: Ако желаем да проверим дали даден обект има някакво свойство, то първо проверяваме дали то е зададено в явен вид (напр. “2 крака” при Хора) и в случай, че то не е явно указано, правим проверка на всички класове, които наследява даденият обект (в случая бихме проверили за броя на крака в класа Бозайници). Наследяването се представя графично много лесно в семантичните мрежи. Представянето обаче се усложнява, когато имаме случай на множествено наследяване, т.е. когато алгоритъмът открие две или повече стойности, които отговарят на заявката му. Поради тази причина в някои обектно-ориентирани езици като Java наследяването от този вид е забранено.

### 9.5.2 Понятийни графи

Понятийните графи са вид формализъм за ПИЗ, който спада към семантичните мрежи. Въведен е от Джон Сова през 1976г., като идеята му е да представя формули от предикатната логика от първи ред чрез графи. Всеки граф представя дадено твърдение, като върховете на графа могат да бъдат както явни обекти (напр. човек, куче), така и абстрактни (напр. гняв, тъга). Дългите при този вид графи не са именувани, вместо това са въведени върхове, които изразяват понятийни връзки. Основно предимство на този подход е, че връзката между много обекти се представя лесно чрез обект от тип връзка.

Пример за концептуален граф:



### 9.5.3 Предимства и недостатъци на семантичните мрежи

Основно предимство на този формализъм е, че графиките се представят много лесно в паметта на компютъра. Семантичните мрежи се представят чрез графи, което означава, че всички алгоритми от теорията на графиките могат да бъдат използвани за решаване на задачи със семантични мрежи. Освен това семантичните мрежи използват голяма част от естествения език за представяне на знания, при това ако две изречения с еднакви значения бъдат представени с график, то графиките им също ще са еднакви. Техни съществени предимства са простотата им, естествеността, нагледността и яснотата на представянето.

Основен проблем на този формализъм е, че семантиката му понякога е неясно дефинирана. Значенията, присвоени на върховете, могат да бъдат двусмислени. Недостатък спрямо предикатната логика от първи ред е, че връзките между различните обекти в семантичните мрежи са единствено бинарни (т.е. могат да се изразяват само бинарни взаимоотношения между обектите). Това означава, че изречения от вида: Fly(Shankar, NewYork, NewDelhi,

Yesterday) не могат да бъдат директно вкарани в семантична мрежа. Специфичен недостатък е и че наследяването, извършването на различни операции и самото реализиране на такива системи се управляват трудно.

## 9.6 Продукционни правила

Представянето на знания чрез системи от продукционни правила спада към декларативния тип формализми за ПИЗ. Характерно за този формализъм е, че знанието се представя чрез двойки от вида: <условие; следствие>. Тези двойки образуват множество от правила, определящи действието на системата. За всяка такава двойка агентът проверява дали условието е издържано и ако е така, продукционното правило задейства следствието и то се изпълнява.

### 9.6.1 Системи, основани на правила

Всяка система с изкуствен интелект, основана на правила има три основни компонента:

1. Работна памет (контекст) – съдържа входните данни за задачата. Те могат да бъдат получени както от потребителя, така и от интерпретатора в процеса на логическия извод. Представянето на данните в работната памет зависи от синтаксиса на правилата, но обикновено е под формата на тройки от вида: <обект;атрибут;стойност> (обектът има атрибут със зададената стойност) или <атрибут;релация;стойност> (атрибутът се намира в дадена релация с конкретната стойност).
2. База от правила – съвкупността от правилата, които са ни дадени за конкретната предметна област. Тук се пази относително постоянната част от данните.
3. Интерпретатор (машината за извод на съответната система) – това е програмна система, чиято основна цел е да прилага въведените в базата правила върху данните от работната памет и да изведе ново знание. Работата на интерпретатора се разделя на две части – избор на правило и изпълнение на правилото. Интерпретаторът може да реализира един от следните два вида алгоритми: алгоритъм за прав или за обратен извод. Правият извод се нарича още извод, управляван от данните, а обратният – извод, управляван от целите.

Прав извод – при този подход интерпретаторът търси правило, чието условие се удовлетворява от данните в контекста, след което изпълнява следствието от двойката, чрез която е представено правилото. Ако съществува обаче повече от едно правило, което удовлетворява даденото условие може да възникне конфликт. Тогава всички, удовлетворяващи условието правила, образуват конфликтно множество и целта ни е да изберем едно от тях, което да изпълним, т.е. прилагаме т.нар. конфликтна резолюция. Има различни подходи за избора на правило – например първото срещнато правило, остойностяване на правилата с тегла или приоритети или правило, което не е използвано до момента и др.

Обратен извод – при този подход интерпретаторът извършва търсене върху десните страни на правилата (т.е. върху следствията). След това се прави проверка дали условието в лявата част на правилото удовлетворява зададената цел. Това се прави, докато се достигне зададената цел или докато бъдат изследвани всички възможности.

### **9.6.2 Пример на система от продукционни правила**

Примерът показва множество от продукционни правила за намиране на огледалния образ на даден низ над азбука, която не съдържа символите: ”\$” и ”\*”, които използваме като специални символи. Базата ни от правила съдържа следните записи:

- P1: \$\$ -> \*
- P2: \*\$ -> \*
- P3: \*x -> x\*
- P4: \* -> null & halt
- P5: \$xy -> y\$x
- P6: null -> \$

В този пример избираме правилото, което да изследване по последователността на правилата в базата. За всяко правило изследваме низа от ляво надясно, докато не съвпадне с лявата страна на някое правило. Ако открием съвпадение, заменяме изходния низ с този от дясната страна на правилото. В нашия пример x и у са променливи, които отговарят на всеки символ от нашата азбука. Нека входният ни низ е ”ABC”, тогава той преминава през следната последователност от трансформации:

\$ABC (P6)  
B\$AC (P5)  
BC\$A (P5)  
\$BC\$A (P6)  
C\$B\$A (P5)  
\$C\$B\$A (P6)  
\$\$C\$B\$A (P6)  
\*C\$B\$A (P1)  
C\*\$B\$A (P3)  
C\*B\$A (P2)  
CB\*\$A (P3)  
CB\*A (P2)  
CBA\* (P3)  
CBA (P4)

Можем да отбележим, че при подобни продукционни системи редът на правилата е от значение за това колко стъпки ще изгълним до достигане на целта.

### **9.6.3 Предимства и недостатъци на продукционните правила**

Основно предимство на този формализъм е естествеността на представяне на знанията. Друго предимство е, че при този формализъм има ясно изразена модулност – т.е. постоянните знания са отделени от временните (първите се пазят в базата от правила, докато вторите са в работната памет). Тази модулност води до лесно изменение на правила в системата. Освен това лесно се откриват и отстраняват грешки в базата от правила, заради структурата на всяко правило – две независими части (условие и следствие). Ако правилото се изпълнява в неподходящия момент, лесно се открива условието, поради което това се случва, а ако се изпълнява неподходящо действие, лесно се открива правилото, чието следствие е даденото действие.

Основен недостатък на продукционните правила като формализъм за ПИЗ е проблемът за обясняването на взетите решения. За разлика от човека, който лесно дава обяснение защо е избрал дадена хипотеза и след това друга, при системите с продукционни правила това не изглежда по този начин. За преодоляване на този проблем е необходимо разширяване на съществуващата система от правила или използване на изцяло друг подход. Друг недостатък е и ниската ефективност при избор на правило от конфликтното множество в случай на възникване на конфликт, както и при работа с големи системи. Немаловажен недостатък е и че не всички знания могат да бъдат представени под формата на правила.

## 9.7 Фреймове

Фреймовете са формализъм за ПИЗ, който се базира върху идеята за представянето на знания в човешкия мозък, а именно като обособени знания около дадено понятие. За първи път тази идея за представяне на знания се въвежда от Марвин Мински през 1975г. Той дефинира понятието фрейм (рамка) като „структурата от данни, предназначена за описание на дадена стереотипна, стандартна ситуация“.

Основна характеристика на фреймовете е, че описват знания, които винаги са верни и структурата на тези знания и връзките между отделните такива е заложена в самия фрейм. Накратко: фреймът съдържа информация за това, което би се случило в следващия момент и какво да се направи, ако това се случи или не се случи. Пример (Мински): „Преди да влезе в дадена стая, човек вече е подготвил в съзнанието си образа на стая „в общи линии“ – стени, таван, под, врата, прозорци и т.н. Когато влезе в стаята, той само допълва и конкретизира своята представа за стая с особеностите на конкретната стая.“ На фреймовете може да се гледа и като на образци с определени особености, които конкретизираме за съответната ситуация.

### 9.7.1 Представяне на знания чрез фреймове

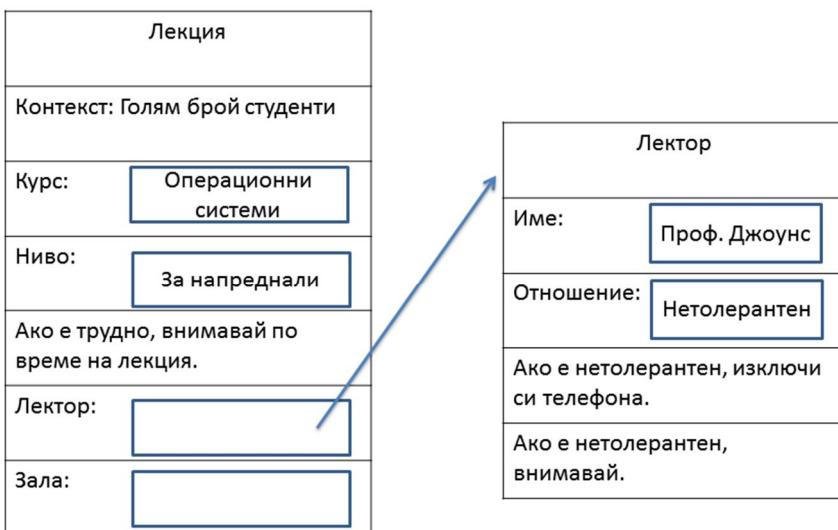
Представянето на знания в дадена система се представя чрез фреймове, като се използва идеята за йерархия от фреймове, която е подобна на тази за йерархия от класове в обектно-ориентираното програмиране. Всеки фрейм се състои от следните елементи:

- Име;
- Слотове - това са свойствата на описвания обект, като свойствата носят и стойностите, които заемат. Всяка стойност може да бъде: стойност по подразбиране; наследена стойност от фрейм на по-високо ниво в йерархијата; процедура, наречена демон, която служи за намиране на стойността; специфична стойност, която може да служи за представяне на изключение от типичните за този слот стойности. Стойностите във фреймовете на по-високо ниво могат да бъдат интервали от стойности, а също и условия. На по-ниските нива стойностите могат да бъдат конкретни стойности, които да пренаписват стойностите, наследени от по-високи нива.

Слотовете във фреймовете могат да съдържат следната информация: информация за избора на фрейм в дадена ситуация; информация за връзките на този фрейм с останалите; процедури, които да се изпълнят след като определени слотове са запълнени; информация по подразбиране, която да се използва в случай на липсванца (непопълнена) информация; други фреймове (така се реализира йерархичната структура).

Важно е да се отбележи, че система с фреймове може да поддържа множествено наследяване, но в такива случаи, тя трябва да предоставя възможност за справяне със случаи на конфликти.

Пример за фрейм:



### 9.7.2 Предимства и недостатъци на фреймовете

Основно предимство на този вид формализъм е естествеността на представянето, тъй като то се доближава до това в човешкия мозък. Модулността, йерархичната структура при представянето им и възможността за задаване на стойности по подразбиране правят фреймовете предпочитан формализъм за ПИЗ. Чрез използването на фреймове за ПИЗ създаването на процедури, въвеждането на стойности по подразбиране и откриването на липсващи стойности се улеснява значително.

Към основните недостатъци на този формализъм спадат неясната семантика, т.к. знанията, представени чрез фреймове, често могат да бъдат тълкувани по различен начин; недостатъчната изразителност на формализма в сравнение с предикатната логика от първи ред. Проблеми възникват и при управлението на наследяването на свойства при представянето чрез фреймове. Фреймовете са трудно програмирани и възникват проблеми при извода на нови знания.

## 9.8 Характеристики на добрия формализъм за ПИЗ

Правилното представяне на знание покрива следните характеристики:

- Добро покритие – представянето на знания да покрива достатъчно добре в ширина и дълбочина необходимата информация за дадена предметна област; Без достатъчно широко покритие на използвания формализъм системата не би била способна да изведе допълнителни знания.
- Разбираемост –добре е чрез използвания формализъм да можем лесно и интуитивно да превеждаме знанията от естествения език на езика на логиката и обратно. Колкото повече знанието е разбито по модуларен или йерархичен

начин, толкова по-лесно ще се комбинира в по-сложни форми и ще доведе до по-лесно и разбираемо извлечане на нови знания.

- Консистентност – добрият формализъм за представяне и използване на знания избягва двусмислици и конфликти между знания.
- Ефективност.
- Леснота при модифициране и обновяване на знания.

## 9.9 Речник

Inference rule	Правило за извод
Connective	Съждителна връзка
Proposition	Съждение, твърдение
Fuzzy	Размит
Inverse link	Обратна връзка
Conceptual graph	Концептуален граф
Forward Chaining	Прав извод
Backward Chaining	Обратен извод

## 9.10 Ресурси

Artificial Intelligence: A Modern Approach (2nd Edition) Stuart Russell, Peter Norvig, 2002

<http://www.mdx.ac.uk/> - Knowledge representation lecture

<http://www.jfsowa.com/cg/>

[http://en.wikipedia.org/wiki/Production\\_system](http://en.wikipedia.org/wiki/Production_system)

Изкуствен интелект, М. Нишева, Д. Шипков,  
изд. „Интеграл“ Добрич, 1995

[http://en.wikipedia.org/wiki/Knowledge\\_representation\\_and\\_reasoning](http://en.wikipedia.org/wiki/Knowledge_representation_and_reasoning)

[http://en.wikipedia.org/wiki/Modal\\_logic](http://en.wikipedia.org/wiki/Modal_logic)

[http://en.wikipedia.org/wiki/Fuzzy\\_logic](http://en.wikipedia.org/wiki/Fuzzy_logic)

# **10 Планиране на действията. Частично наредени планове.**

В планирането имаме възможност да видим как един агент може да се възползва от структурата на проблема за конструиране на сложни планове за действие.

## **Съдържание**

[Съдържание](#)

[Дефиниция на планиране](#)

[Езици за планиране на проблеми](#)

[Особености на езика](#)

[Изразителност и разширения](#)

[Примери](#)

[Планиране с търсене в пространството от състояния](#)

[Алгоритъм за планиране чрез прогресия](#)

[Алгоритъм за планиране чрез регресия](#)

[Евристики за търсене в пространството от състояния](#)

[Частично наредени планове](#)

[Частично наредените планове като задача за търсене](#)

[Примери за частично наредени планове](#)

[Планиране на действията. Частично наредени планове | Речник](#)

[Планиране на действията. Частично наредени планове | Ресурси](#)

## **10.1 Дефиниция на планиране**

Подход, при който е нужно генерирането на последователност от действия за изпълнение на задача или за постигане на определена цел, се нарича **планиране**.

За целта на темата ще разглеждаме само среди, които са напълно обозрими, определени, крайни, статични (наблюдава се промяна само тогава, когато агентът извършва някакво действие) и дискретни. За по-кратко тези среди ще наричаме **класически среди за планиране**.

Нека да разгледаме какво се случва, ако един обикновен агент използва стандартни алгоритми за търсене – напр. търсене в дълбочина, A\* и т.н. – за разрешаване на проблеми от реалния свят. Това ще ни помогне да създадем по-добри агенти за планиране. Най-очевидната пречка е, че агентът може да бъде затруднен от предприемането неподходящи действия.

*Проблем:* Кои действия са подходящи?

Следващият проблем, който възниква, е намирането на добра **евристична функция**(евристика).

*Проблем:* Коя функция е добра евристика?

И накрая, решението на проблема може да се окаже неефективно, т.к. не може да се използва предимството от декомпозирането на проблема на по-малки проблеми.

*Проблем:* Как да декомпозираме даден проблем на части?

## 10.2 Езици за планиране на проблеми

Ще разглеждаме представянето на даден проблем като съвкупност от **състояния, действия и цели**. Нашата цел е да намерим език, който:

- е достатъчно изразителен, за да може да описва широк спектър от проблеми
- е достатъчно строг, за да позволи създаването на ефективни алгоритми
- да се възползва от логическата структура на проблема

В този раздел, ще очертаем основното представяне на език за класическо планиране, познат като **STRIPS**(**STanford Research Institute Problem Solver**). По-късно ще посочим и някои от разширенията на STRIPS-подобни езици.

### 10.2.1 Особености на езика

#### Представяне на състоянията.

Планирането декомпозира света до набор от логически условия и представя едно състояние като конюнкция на положителни литерали.

Литерали за твърдение: *Poor* ^ *Unknown* – може да изобразява състоянието на злополучен агент.

Литерали от първи ред(основни и функционално-свободни): *At (Plane1, Melbourne)* ^ *At (Plane2, Sydney)*.

Литерали като *At (x, y)* или *At (Father (Fred), Sydney)* не са позволени.

Приема се, че светът е затворен - това означава, че всички условия, които не се срецват в състояние се считат за лъжа.

Представяне на целите – могат да бъдат представени като: Частично определено състояние, представено като конюнкция на положителни основни литерали(напр. *Rich* ^ *Famous* или *At (P2, Tahiti)*).

Целта е достигната, ако състоянието съдържа всички литерали на целта или казано по друг начин - състоянието *s* удовлетворява целта *g*, ако *s* съдържа всички атоми на *g*(други също са възможни). Например състоянието *Rich* ^ *Famous* ^ *Miserable* изпълнява целта *Rich* ^ *Famous*.

#### Представяне на действията.

Действието се специфицира в изразите на предусловията, които трябва да притежава, преди да може да бъде изпълнено, и ефектите, които настъпват по време на неговото изпълнение, т.e. *Действие* = *Предусловие* + *Ефект*. Например, действието на летене със самолет от едно място до друго, можем да опишем по следния начин:

*Action(Fly(p, from, to),*

*PRECOND: At(p, from)  $\wedge$  Plane(p)  $\wedge$  Airport(from)  $\wedge$  Airport(to)*

*EFFECT:  $\neg$ At(p, from)  $\wedge$  At(p, to))*

По-правило е да наричаме това **схема на действие** - тя представя няколко различни действия, които могат да произлязат при инстанцирането на променливите *p*, *from* и *to* към различни константи. Като цяло схемата на действие се състои от следните три части:

Името на действието и списъка с параметри – напр. *Fly(p, from, to)* служи за идентификация на действието.

**Предусловието** е конюнкция на функционално-свободни положителни литерали, посочващи какво трябва да бъде изпълнено в състоянието преди да може да се изпълни действието. Всички променливи в предусловието трябва да се срещат в списъка с параметри на действието.

**Ефектът** е конюнкция на функционално-свободни литерали, описващи как се изменя състоянието, когато действието бъде изпълнено. За положителния литерал *P* в ефекта, може да се твърди, че е истина в резултат на действие, тогава когато за негативният литерал  $\neg P$  се, твърди че е лъжа. Променливите на ефекта също трябва да се срещат в списъка с параметри на действието.

За да се подобри яснотата, някои системи за планиране разделят ефекта в списъци за добавяне за положителните литерали и в списъци за изтриване за отрицателните.

Имайки дефиниран синтаксис за представяне на проблемите при планиране, сега трябва да дефинираме семантиката. Най-простият начин да направим това е да опишем как действията оказват влияние върху състоянията.

*Проблем:* Как действията влияят върху състоянията?

Първо, ще казваме, че действието е **приложимо** във всяко състояние, което отговаря на предусловието. В противен случай действието няма да има никакъв ефект. За схемата на действие от първи ред, установяването на приложимост ще включва замяна на променливите в предусловието. Например да предположим, че текущото състояние се описва от:

*At(P<sub>1</sub>, JFK)  $\wedge$  At(P<sub>2</sub>, SFO)  $\wedge$  Plane(P<sub>1</sub>)  $\wedge$  Plane(P<sub>2</sub>)  
 $\wedge$  Airport(JFK)  $\wedge$  Airport(SFO) .*

Това състояние удовлетворява предусловието

*At(p, from)  $\wedge$  Plane(p)  $\wedge$  Airport(from)  $\wedge$  Airport(to)*

със замяна  $\{p/P_1, \text{from}/\text{JFK}, \text{to}/\text{SFO}\}$ . По този начин действието *Fly(P<sub>1</sub>, JFK, SFO)* е приложимо.

**Резултатът** от изпълнението на действието  $a$  в състоянието  $s$  е състояние  $s'$ .  $s'$  е същото като  $s$  освен, ако:

- Всеки положителен литерал  $P$  в ефекта е прибавен в  $s'$
- Всеки отрицателен литерал  $\neg P$  е премахнат от  $s'$

$At(P1, SFO) \wedge At(P2, SFO) \wedge Plane(P1) \wedge Plane(P2) \wedge Airport(JFK) \wedge Airport(SFO)$

Тази дефиниция се въплъща в така нареченото STRIPS-допускане, където всеки литерал, който не се среща в ефекта, остава непроменен.

Най накрая, можем да дефинираме решение на проблема за планиране. В най-простата си форма, това е просто последователност от действия, които биват изпълнявани в начално състояние, резултатът на което е състояние удовлетворяващо целта.

### 10.2.2 Изразителност и разширения

Различни ограничения, наложени от реализацията на STRIPS, са избрани с надеждата да направят планиращите алгоритми по-прости и по-ефективни, без да се има предвид, че това може да го направи твърде труден за описание на проблемите от реалния свят. Едно от най-важните ограничения е това, че литералите са функционално-свободни. Чрез това ограничение, можем да бъдем сигурни, че всяка схема на действие за даден проблем може да се опише теоретично. Ако позволим функционалните символи, тогава ще има възможност да се конструират безкрайно много състояния и действия.

В последните години стана ясно, че STRIPS не е достатъчно изразителен за някои реални ситуации. В резултат на това бяха разработени много различни варианти на езици. Следващата таблица описва накратко едно важно разширение – **Action Description Language – ADL**, като го сравнява със STRIPS.

STRIPS	ADL
Само положителни литерали в състоянието: $Poor \wedge Unknown$	Положителни и отрицателни литерали в състоянието: $\neg Rich \wedge \neg Famous$
Допускане за затворен свят – неспоменатите литерали са лъжа.	Допускане за отворен свят – неспоменатите литерали са неизвестни.
Ефектът $P \wedge \neg Q$ означава добави $P$ и изтрий $Q$ .	Ефектът $P \wedge \neg Q$ означава добави $P$ и $\neg Q$ и изтрий $\neg P$ и $Q$ .
Само основни литерали в целите: $Rich \wedge Famous$	Променливи с квантор в целите: $\exists x At(P_1, x) \wedge At(P_2, x)$ е целта да има $P_1$ и $P_2$ на едно и също място.
Целите са конюнкции:	Целите са конюнкции и дизюнкции:

$Rich \wedge Famous$	$\neg Poor \wedge (Famous \vee Smart)$
Ефектите са конюнкции.	Позволени са условните ефекти: <i>when P: E</i> - означава <i>E</i> е ефект, само ако <i>P</i> е удовлетворено.
Липсва поддръжка на сравнение.	Предиката за сравнение е вграден, напр. ( $x = y$ )
Липсва поддръжка на типове.	Променливите могат да имат типове, напр. ( $p : Plane$ ).

Таблица 1: Сравнение на езиците за планиране - STRIPS и ADL. И в двата случая целите функционират като предусловие за действие без параметри.

В езика ADL действието Fly може да бъде представено по следния начин:

*Action(Fly( $p : Plane$ , from : Airport, to : Airport)),*  
*PRECOND:At( $p$ , from)  $\wedge$  (from  $\neq$  to)*  
*EFFECT: $\neg$ At( $p$ , from)  $\wedge$  At( $p$ , to)) .*

Съществува и друг вариант на език за планиране – **PDDL**(Planning Domain Definition Language). Този език позволява на изследователите да обменят benchmark тестове и да сравняват резултати.

### 10.2.3 Примери

Air cargo transport – задача за въздушен превоз на товари

```

Init(At( $C_1$ , SFO)  $\wedge$  At( $C_2$ , JFK)  $\wedge$  At( $P_1$ , SFO)  $\wedge$  At( $P_2$ , JFK)
      $\wedge$  Cargo( $C_1$ )  $\wedge$  Cargo( $C_2$ )  $\wedge$  Plane( $P_1$ )  $\wedge$  Plane( $P_2$ )
      $\wedge$  Airport(JFK)  $\wedge$  Airport(SFO))
Goal(At( $C_1$ , JFK)  $\wedge$  At( $C_2$ , SFO))
Action(Load( $c$ ,  $p$ ,  $a$ ),
       PRECOND: At( $c$ ,  $a$ )  $\wedge$  At( $p$ ,  $a$ )  $\wedge$  Cargo( $c$ )  $\wedge$  Plane( $p$ )  $\wedge$  Airport( $a$ )
       EFFECT:  $\neg$ At( $c$ ,  $a$ )  $\wedge$  In( $c$ ,  $p$ ))
Action(Unload( $c$ ,  $p$ ,  $a$ ),
       PRECOND: In( $c$ ,  $p$ )  $\wedge$  At( $p$ ,  $a$ )  $\wedge$  Cargo( $c$ )  $\wedge$  Plane( $p$ )  $\wedge$  Airport( $a$ )
       EFFECT: At( $c$ ,  $a$ )  $\wedge$   $\neg$ In( $c$ ,  $p$ ))
Action(Fly( $p$ , from, to),
       PRECOND: At( $p$ , from)  $\wedge$  Plane( $p$ )  $\wedge$  Airport(from)  $\wedge$  Airport(to)
       EFFECT:  $\neg$ At( $p$ , from)  $\wedge$  At( $p$ , to))
```

Горният фрагмент ни демонстрира проблема за въздушен превоз на товари чрез товарене и разтоварване на и от самолети и техният превоз от едно място до друго. Проблемът може да се дефинира чрез следните 3 действия:

- товарене - *Load*
- разтоварване - *Unload*

- превоз - *Fly*

Действията оказват влияние върху два предиката:

- *In (c, p)* - товарът *c* е натоварен в самолета *p*
- *At (x, a)* - обектът *x*(или самолет, или товар) се намира на летище *a*

Обърнете внимание, че товарът не е навсякъде(*At*), когато е в самолета(*In*), така че *At* наистина означава „на разположение за използване за дадена дестинация“. Необходимо е известно време за запознаване с дефинициите на действията, за да се вникне задълбочено в детайлите. Следният план е решение на проблема:

*[Load(C<sub>1</sub>, P<sub>1</sub>, SFO), Fly(P<sub>1</sub>, SFO, JFK),  
Load(C<sub>2</sub>, P<sub>2</sub>, JFK), Fly(P<sub>2</sub>, JFK, SFO)]*.

Нашето представяне на проблема използва езикът STRIPS. По-специално – той позволява на самолета да лети до и от едно и също летище. За сметка на това, чрез използване на литерал за неравно, ADL може да предотврати този проблем.

Spare tire problem – проблем за резервната гума

Да разгледаме проблема за смяна на спукана гума. По-точно, целта ни е да има добра резервна гума, правилно монтирана върху оста на колата, където началното състояние е “има спукана гума и добра резервна гума в багажника”. За да опростим нашата версия на проблема ще разгледаме единствено следните 4 действия:

- отстраняване на резервната гума от багажника
- премахване на спуканата гума от колата
- поставяне на резервната гума на колата
- оставяне на колата без надзор през нощта

Предполагаме, че в квартала, в който сме оставили колата, се случват много кражби и ефектът да я оставим за една нощ без надзор е гумите да изчезнат. За решението на задачата ще използваме ADL.

```

Init(At(Flat, Axle) ∧ At(Spare, Trunk))
Goal(At(Spare, Axle))
Action(Remove(Spare, Trunk),
    PRECOND: At(Spare, Trunk)
    EFFECT: ¬ At(Spare, Trunk) ∧ At(Spare, Ground))
Action(Remove(Flat, Axle),
    PRECOND: At(Flat, Axle)
    EFFECT: ¬ At(Flat, Axle) ∧ At(Flat, Ground))
Action(PutOn(Spare, Axle),
    PRECOND: At(Spare, Ground) ∧ ¬ At(Flat, Axle)
    EFFECT: ¬ At(Spare, Ground) ∧ At(Spare, Axle))
Action(LeaveOvernight,
    PRECOND:
    EFFECT: ¬ At(Spare, Ground) ∧ ¬ At(Spare, Axle) ∧ ¬ At(Spare, Trunk)
           ∧ ¬ At(Flat, Ground) ∧ ¬ At(Flat, Axle))

```

Blocks world – задача за света на кубовете

Една от най-известните задачи на планирането е познатата като **светът на кубовете**.

```

Init(On(A, Table) ∧ On(B, Table) ∧ On(C, Table) ∧ Block(A) ∧ Block(B) ∧ Block(C) ∧ Clear(A) ∧
      Clear(B) ∧ Clear(C))
Goal(On(A, B) ∧ On(B, C))
Action(Move(b, x, y)
    PRECOND: On(b, x) ∧ Clear(b) ∧ Clear(y) ∧ Block(b) ∧ (b ≠ x) ∧ (b ≠ y) ∧ (x ≠ y)
    EFFECT: On(b, y) ∧ Clear(x) ∧ ¬ On(b, x) ∧ ¬ Clear(y))
Action(MoveToTable(b, x)
    PRECOND: On(b, x)           ∧ Clear(b) ∧ Block(b) ∧ (b ≠ x)
    EFFECT: On(b, Table) ∧ Clear(x) ∧ ¬ On(b, x))

```

Едно възможно решение може да разгледаме чрез следния фрагмент код:

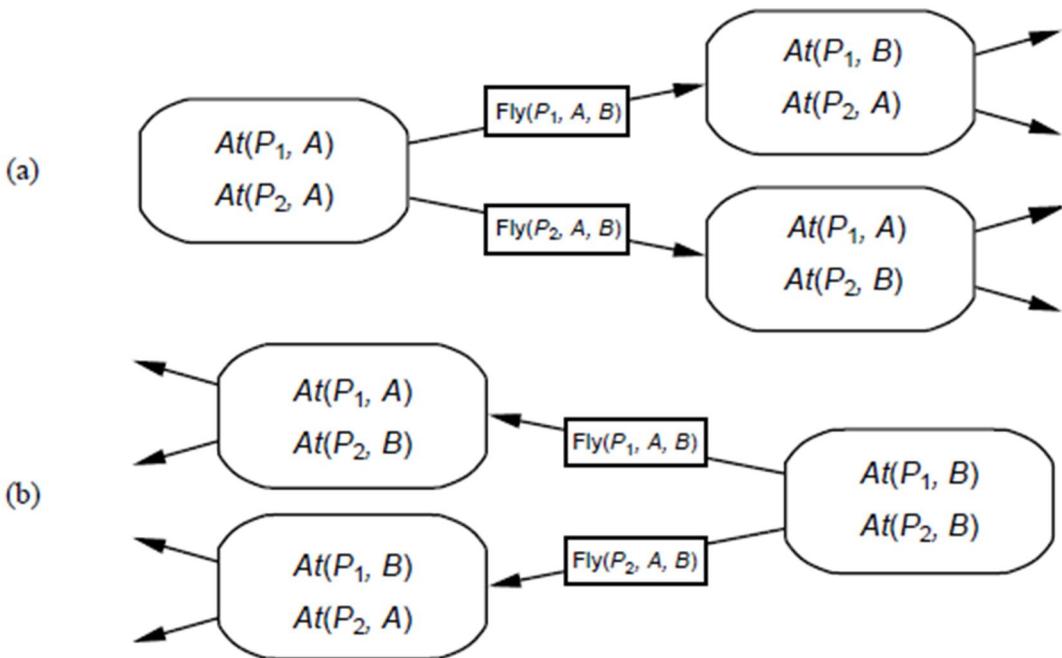
[Move(B, Table, C), Move(A, Table, B)].

### 10.3 Планиране с търсене в пространството от състояния

Сега ще насочим вниманието си към планирането на алгоритми. Най-разбираемият подход е като използваме търсене в пространството от състояния. Тъй като описанията на действията определят както предусловията, така и ефектите, е възможно търсенето да се осъществи в една от двете посоки:

Планиране чрез прогресия - търсене напред от началното състояние. Разглежда ефекта от всички възможни действия в дадено състояние. – *фиг. (a)*

Планиране чрез ргресия - търсене назад от целта. За достигане на целта, това което трябва да бъде вярно е предишното състояние. – *фиг. (b)*



Също така може да използваме предварително формулирано действие и представяне на целта, за да получим ефективна евристика.

### 10.3.1 Алгоритъм за планиране чрез прогресия

#### **Формулировка на проблема.**

За **начално състояние** при търсенето ще считаме началното състояние на проблема. Като цяло, всяко състояние е множество от положителни основни литерали. Литерали, които не се споменават са лъжа.

**Действията**, които са приложими в състоянието, са всички онези чиито предусловия са изпълнени – на принципа: добавяне на положителни ефекти и изтриване на отрицателни.

**Тестът на целта** проверява дали състоянието удовлетворява целта на проблема.

**Цената на стъпката** на всяко действие обикновено е 1.

Да си припомним, че в отсъствието на функционални символи, пространството от състояния на проблема е крайно множество. Ето защо всеки алгоритъм за търсене в графи е пълен, напр.  $A^*$ .

Недостатъци на алгоритъма:

- Проблемът с предприемането на погрешно действие.
- Добрата евристика е задължителна за постигането на ефективно търсене.

### 10.3.2 Алгоритъм за планиране чрез регресия

Въпросите, които изникват, са: как да определим предшествениците и кои са състоянията, които след прилагане на някакво действие биха водили до целта?

**Целево състояние:**  $At(C_1, B) \wedge At(C_2, B) \wedge \dots \wedge At(C_{20}, B)$

Подходящо действие за първата конюнкция:  $\text{Unload}(C_1, p, B)$ ; Ще работи, само ако предусловията са изпълнени.

Определяме предишното състояние:  $\text{In}(C_1, p) \wedge \text{At}(p, B) \wedge \text{At}(C_2, B) \wedge \dots \wedge \text{At}(C_{20}, B)$

Субцелта  $\text{At}(C_1, B)$  не трябва да бъде в това състояние.

В допълнение към изискването на действията да достигнат до някакъв желан литерал, трябва да се уверим, че тези действия не премахват всички желани литерали. Действие, което удовлетворява тези ограничения, се нарича **консистентно** или **последователно**.

Главното предимство на този алгоритъм е, че предприетите действия са винаги уместни. Често факторът на разклонение е много по-малък, отколкото при алгоритъма за планиране чрез прогресия.

Като имаме предвид дадените дефиниции, можем да опишем общия процес за намиране на предшественици при търсенето чрез регресия. Като се има предвид описането на целта  $G$ , нека  $A$  е действие, което е уместно и консистентно. Тогава съответният предшественик е както следва:

Всички положителни ефекти описани в  $A$ , които се срещат в  $G$  се премахват.

Ако  $A$  включва предусловни литерали, неописани в  $G$ , ги добавяме, т.е. всеки предусловен литерал на  $A$  се добавя, освен ако вече не се среща.

Търсенето може да се осъществи чрез всеки стандартен алгоритъм за търсене. Прекратяваме процеса, когато е генерирано описание на предшественик, което се удовлетворява от началното състояние на проблема.

### 10.3.3 Евристики за търсене в пространството от състояния

Нито един от разгледаните по-горе алгоритми не е достатъчно ефективен без добра евристична функция. Основната идея е да се изчисли колко действия биха били необходими до достижането на целта. Намирането на точният брой действия е NP-трудна задача, но е възможно да се пресметнат разумни приближения, без прекалено много изчисления. Ще разгледаме два подхода, които могат да бъдат използвани.

Първият е да се опитаме да извлечем по-слаб проблем от дадената ни спецификация на проблем. Оптималното решение на по-слабия проблем – до което, вярваме, се достига по-лесно – ни дава допустима евристика за първоначалния проблем.

Вторият подход е да предположим, че класическият алгоритъм „разделяй и владей“ ще работи. Това се нарича предположение за **субцелева независимост**: цената за решаване на конюнкциите на субцелите е приблизително сумата от цената на решението на всеки независим малък проблем.

## 10.4 Частично наредени планове

Досега разгледаните планирания (прогресивно и регресивно) са видове пълно наредени планирания. Можем да изградим пълно нареден план, работейки с частично наредени

планове, напр. задача за обуване на обувки и чорапи. Планът, който изграждаме, включва две независими последователности от действия - за ляв и десен крак. За разлика от пълните наредби, тук не заявяваме строг ред на изпълнение на тези последователности.

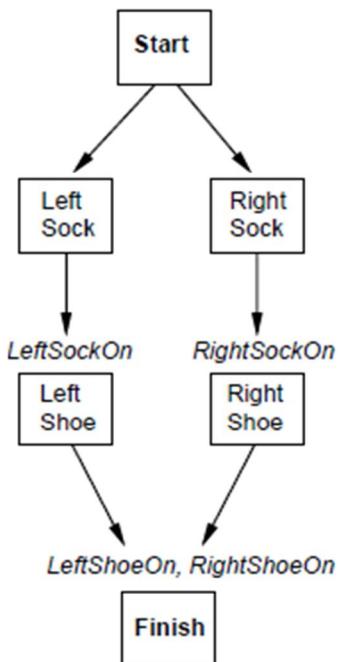
```

Goal(RightShoeOn ∧ LeftShoeOn)
Init()
Action(RightShoe, PRECOND:RightSockOn, EFFECT:RightShoeOn)
Action(RightSock, EFFECT:RightSockOn)
Action(LeftShoe, PRECOND:LeftSockOn, EFFECT:LeftShoeOn)
Action(LeftSock, EFFECT:LeftSockOn) .

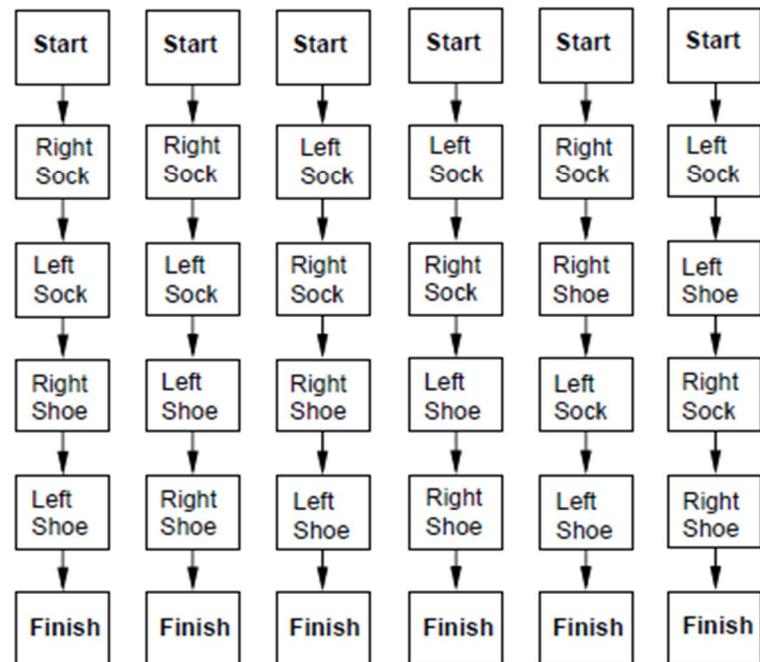
```

Всеки алгоритъм за планиране, в който могат да се поставят две действия в план, без да се уточнява кое се случва първо, се нарича **алгоритъм за частично наредени планове**. Следващата фигура показва частично нареден план, който е решение на задачата за обувките и чорапите. Имайте предвид, че решението е представено във вид на графика на действията, а не като таката тяхна последователност.

Partial-Order Plan:



Total-Order Plans:



#### 10.4.1 Частично наредените планове като задача за търсене

Състоянията са, най-често незавършени, планове. Празният план съдържа единствено началното и финалното действие. Всеки план се състои от 4 компоненти:

- Множество от **действия** – това са стъпките на плана.
- Множество от **наредени ограничения**:  $A < B$ . Циклите представляват противоречия.
- Множество от **причинно-следствени връзки**:  $A \dashv p \dashv B$

- Множество от **отворени предусловия** – ако предусловието не е достигнато от някое действие на плана.

С частично нареденият план действаме по следния начин:

1. Избираме едно отворено предусловие.
2. Търсим всички консистентни планове-наследници, т.е. поредици от действия, които удовлетворяват условието.
3. Добавяме причинно-следствените връзки и ограниченията към плана ( $A \rightarrow p \rightarrow B$  и  $A < B$ ).
4. Разрешаваме конфликтите между новата причинно-следствена връзка и съществуващите планове, както и между новото състояние  $A$  и съществуващите планове.

Един план е **консистентен**, ако в него няма цикли в наредените ограничения, и няма конфликти в причинно-следствените връзки. В процеса на действие, частично нареденият план се преобразува в пълно нареден. Консистентен план, който не съдържа отворени предусловия е решение.

Вече сме готови да формулираме задачата, която частично наредените планове решават. Ще започнем с формулирането на подходящо твърдение за проблемите на планирането. Както обикновено, дефиницията включва началното състояние, действията и целта.

Началният план:

- съдържа състоянията *Начало* и *Край*
- ограничението *Начало*  $<$  *Край*
- не съдържа причинно-следствени връзки
- всички предусловия в *Край* са отворени

Функция на наследника – избира произволно едно отворено предусловие  $p$  на действие  $B$  и генерира план за наследника на всяко възможно съответствие на действието  $A$ , което удовлетворява  $p$ .

Тестването на целта проверява дали планът е решение на първоначално поставения проблем, т.к. са генериирани само консистентни планове, тестването трябва да провери дали не се съдържат отворени предпоставки.

#### **10.4.2 Примери за частично наредени планове**

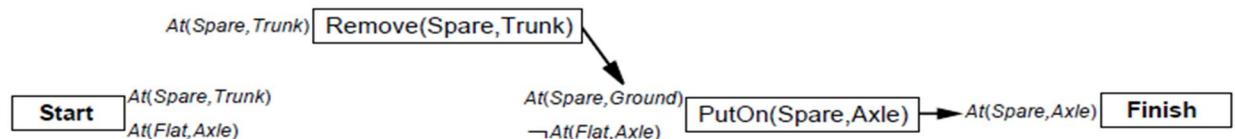
Проблем за резервната гума

```

Init(At(Flat, Axle) ∧ At(Spare, Trunk))
Goal(At(Spare, Axle))
Action(Remove(Spare, Trunk),
    PRECOND: At(Spare, Trunk)
    EFFECT: ¬ At(Spare, Trunk) ∧ At(Spare, Ground))
Action(Remove(Flat, Axle),
    PRECOND: At(Flat, Axle)
    EFFECT: ¬ At(Flat, Axle) ∧ At(Flat, Ground))
Action(PutOn(Spare, Axle),
    PRECOND: At(Spare, Ground) ∧ ¬ At(Flat, Axle)
    EFFECT: ¬ At(Spare, Ground) ∧ At(Spare, Axle))
Action(LeaveOvernight,
    PRECOND:
    EFFECT: ¬ At(Spare, Ground) ∧ ¬ At(Spare, Axle) ∧ ¬ At(Spare, Trunk)
        ∧ ¬ At(Flat, Ground) ∧ ¬ At(Flat, Axle))

```

Решение на проблема:



Инициализация на плана: стартиране с ефектите и завършване с предусловието.

Избираме отворена предпоставка:  $At(Spare, Axle)$ .

Единствено  $PutOn(Spare, Axle)$  е приложимо.

Добавяме причинно-следствената връзка:  $PutOn(Spare, Axle) \rightarrow At(Spare, Axle) \rightarrow Finish$ .

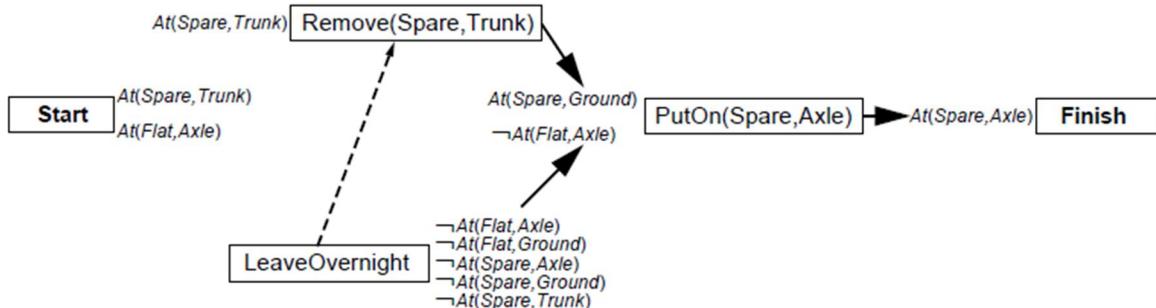
Добавяме ограничението:  $PutOn(Spare, Axle) < Finish$ .

Избираме отворена предпоставка:  $At(Spare, Ground)$ .

Единствено  $Remove(Spare, Trunk)$  е приложимо.

Добавяме причинно-следствената връзка:  $Remove(Spare, Trunk) \rightarrow At(Spare, Ground) \rightarrow PutOn(Spare, Axle) \rightarrow At(Spare, Axle) \rightarrow Finish$ .

Добавяме ограничението:  $Remove(Spare, Trunk) < PutOn(Spare, Axle)$ .



Избираме отворена предпоставка:  $At(Spare, Ground)$ .

$LeaveOverNight$  е приложимо.

Конфликт:  $Remove(Spare, Trunk) \rightarrow At(Spare, Ground)$   $PutOn \rightarrow At(Spare, Axle)$ .

За да разрешим конфликта, добавяме следното ограничение:  $LeaveOverNight < Remove(Spare, Trunk)$ .

Добавяме причинно-следствена връзка:  $LeaveOverNight \rightarrow At(Spare, Ground) \rightarrow PutOn(Spare, Axle)$ .

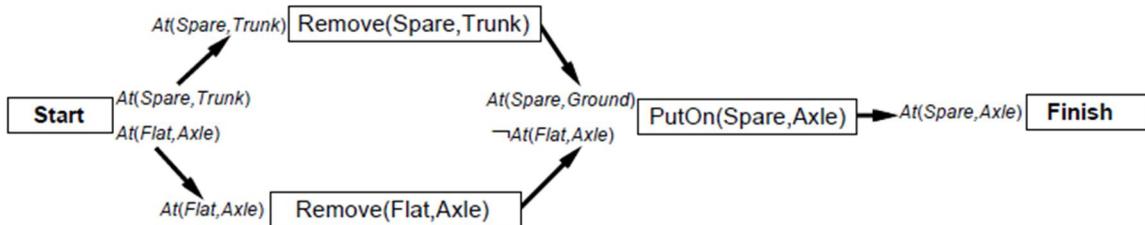
Избираме отворена предпоставка:  $At(Spare, Trunk)$ .

Единствено  $Start$  е приложимо.

Добавяме причинно-следствена връзка:  $Start \rightarrow At(Spare, Trunk) \rightarrow Remove(Spare, Trunk)$ .

Конфликт: на причинно-следствената връзка с ефект  $At(Spare, Trunk)$  в  $LeaveOverNight$ . Не е възможно пренареждане на решението.

Връщане назад.



Премахваме  $LeaveOverNight$ ,  $Remove(Spare, Trunk)$  и причинно-следствените връзки.

Повтаряме стъпката с  $Remove(Spare, Trunk)$ .

Добавяме също  $RemoveFlatAxle$ .

Край.

## 10.5 Речник

classical planning environments	классически среди за планиране
heuristic function	евристика/евристична функция
decomposition problem	декомпозиране на проблем
state	състояние
action	действие
goal	цел

propositional literals	литерали за твърдение
first-order literals	литерали от първи ред
function-free	функционално-независим/функционално-свободен
closed-world assumption	приемане за затворен свят
action schema	схема на действие
precondition	предусловие/предпоставка
effect	ефект
applicable	приложимо
progression planning	планиране чрез прогресия
regression planning	планиране чрез регресия
initial state	начално състояние
goal state	целево състояние/цел
goal test	тест на целта
step cost	цена на стъпка
predecessor	предшественик
consistent	консистентен/последователен
branching factor	фактор на разклонение
relaxed problem	по-слаб проблем
subgoal independence	субцелева независимост
totally ordered	пълно нареден
partial-order plan	частично наредени план
ordering constraints	наредени ограничения
causal links	причинно-следствени връзки
conflicts	конфликти
open preconditions	отворени предусловия

## 10.6 Ресурси

Artificial Intelligence: A Modern Approach (2nd Edition) Stuart Russell, Peter Norvig  
2002

# **11 Разсъждения с несигурни знания. Правило на Бейс.**

## **Условна независимост.**

Извеждането на изводи може да става както в условията на пълна определеност на наличните факти и знания, така и в условията на известна несигурност.

### [Съдържание](#)

[Увод в темата](#)

[Въведение](#)

[Използване на правилата](#)

[Известни подходо](#)

[Размити множества](#)

[Видове изводи](#)

[Дедукция](#)

[Абдукция](#)

[Индукция](#)

[Аналогия](#)

[Вероятности](#)

[Формула на Бейс](#)

[Речник](#)

[Ресурси](#)

### **11.1 Защо са необходими?**

Разсъжденията с несигурни знания са необходими поради ограниченията на класическата логика. Тя позволява изграждането на системи за разсъждене, основани на правила от типа  $p \rightarrow q$ ,  $s \wedge q \rightarrow r$  и т.н. Приема се при това, че са изпълнени следните условия:

1. Всяко съждение е или абсолютна истина, или абсолютна лъжа.
2. Самите правила като цяло представлят истинни съждения.
3. Всички предпоставки (антепреденти) на правилата са истинни.
4. Получените заключения се считат за верни за целия процес на извеждане (не може да бъдат отречени в следващите стъпки).

Само при такава постановка може да се твърди, например, че в правилото  $p \rightarrow q$  (истинността на)  $q$  еднозначно следва от (истинността на)  $p$ .

В реална обстановка, един агент почти никога не разполага с “пълната истина” относно достоверността на знанията, с които оперира. Те могат да бъдат неточни, неясни, неопределени, непълни и т.н.

Например, възможно е агентът да знае, че даден факт е “почти винаги верен”. Също така в процеса на неговото функциониране може да постъпи информация, която противоречи на вече изведената от него до момента.

Знания от този тип наричаме несигурни знания. Източниците за несигурност на знанията може да са най-различни:

- **Ограниченията** на сензорната/изпълнителната система на агента.
- **Свойствата** на средата и некоректното им разбиране от агента.
- **Невъзможността** за указване на всички възможни условия за изпълнение на някакво действие и др.

Поради тези причини, не можем да бъдем напълно сигурни, че в правилото  $p \rightarrow q$ , например, твърдението  $p$  е вярно. При такава несигурност, правилото е неприложимо според законите на формалната логика.

## 11.2 Как да се използва правилото в този случай?

• **Интуитивно**, би могло да се въведе някаква мяра за несигурност, например вероятностна (за която ще говорим по-късно). Тогава, ако се предположи, че твърдението  $p$  е вярно с вероятност например 90%, то би могло да се изиска заключението  $q$  да е вярно със същата вероятност.

• Нека сега разгледаме по-сложното правило  $p \wedge q \rightarrow r$ . Ако вероятността  $p$  да е вярно е 80%, а вероятността  $q$  да е вярно е 50%, каква е вероятността  $r$  да е вярно? Средното от двете вероятности (65%), по-ниската от двете (50%), а може би тяхното произведение (40%). Всеки от тези три варианта има някакъв интуитивен смисъл. На никой от тях, обаче, не може да се разчита, когато в някакво разсъждение се прилага дълга верига от сложни и разнообразни правила, за да се достигне до търсеното заключение.

• Освен това, какво да правим, когато самото правило е **несигурно**? Например, правилото  $p \wedge q \rightarrow r$  може да е вярно само понякога, да речем в 90% от случаите. Такъв тип правило е например следното:

“Ако един пациент има висока температура, той е болен от грип”. Ясно е, че правилото не е валидно във всички възможни случаи, а само в някаква част от тях.

• **Забележете** също така, че в горния случай, дори да обърнем посоката на импликацията, казаното остава в сила. Причината за това се крие във факта, че не съществува (и в двете посоки) причинно следствена връзка (достатъчност) между симптома “наличие на висока температура” и “заболяване от грип”. Разбира се, правила от този тип са полезни, защото са коректни в една значителна част (например 78%) от случаите. Очевидно, за подобни случаи са необходими системи за разсъждение с несигурни знания, които да са математически издържани по начин, аналогичен на формалната логика.

### **11.3 Какви подходи са известни**

По настоящем в ИИ се разграничават две основни групи подходи за представяне и разсъждение с несигурни знания:

**1. Числови подходи**, които използват количествена мяра за несигурност (число от интервала  $[0,1]$ ) в комбинация с чисто логически подходи.

Основни представители на тази група са вероятностният подход и размитият подход.

**2. Знакови подходи**, които използват по-скоро качествена мяра за несигурност на знанията. Типични представители на тази група са различни разширения на класическата логика, известни като некласически логики. Обикновено това са логики с ревизия на знанията (наричат се още немонотонни логики). Те допускат промяна на истинността на вече изведени факти при поява на причини (нови знания или опровержения) за това. Измежду най-разпространените са следните типове:

- **логики по подразбиране** - предоставят техники за формализация на разсъждения с непълни знания; използват правила от вида “при липса на противоречеща информация за X приеми, че X е вярно”;
- **модални логики** - при тях истинността на съставните съждения зависи не само от тяхната структура, но и от т.н. модални оператори, като “възможно”, “необходимо”, “винаги”, “понякога”, “зная”, “вярвам” и др.;
- **автоепистемични логики** – позволяват разсъждения относно собствени (на агента) знания и предположения;
- **системи за поддържане на истинност** - работят с непълни и противоречиви знания.

Ще разгледаме основно подходите от първата група, главно поради повишения интерес към тях. Обсъждат се две много важни, особено за технически приложения системи, които позволяват изграждане на интелектуални агенти за работа с несигурни знания. Първата се опира на теорията на теорията на възможностите (теория на размитите множества), а втората - вероятностите и математическата статистика.

Съществуваше период, в който се смяташе, че теорията на размитите множества е частен случай на теорията на вероятностите, че двете теории се припокриват и т.н. Веднага ще подчертаем, че двете теории отразяват два съвършено различни аспекти на несигурността на информацията - случайност и неясност (размитост).

Независимо, че и двете теории използват като мяра за несигурност числа от интервала  $[0,1]$  случайността и размитостта са концептуално различни.

- **Случайността** е несигурност по отношение на настъпването на едно събитие.
- **Размитостта** е несигурност по отношение на степента, в която се е проявило едно осъществено събитие.

Дали едно събитие ще настъпи е случайност.

В каква степен то се е проявило е размитост.

**Пример:**

а) Изречението “Ако вярът днес не престане, утре с вероятност 90% ще вали дъжд” съдържа несигурност от типа случайност. То отразява вярата, степента на убеденост на субекта, който формира изречението, че събитието ще настъпи. Вероятността (мярата на несигурността) може да бъде определена на основата на статистически данни - относно географското положение, годишния период, влажността на въздуха и т.н., или въз основа на някакви общи правила, собствен опит и др. Важно е да се разбере, че “90% вероятност” не означава “90% истина”, а по скоро 90% убеденост (много високо очакване), че в 90% от случаите, приличащи на наблюдавания случай, (облачност, годишен период и т.н.), изречението е вярно (събитието ще настъпи). Само в 10% от случаите, то е невярно. От онтологична гледна точка, теорията на вероятностите (както и класическата логика) приема, че едно изречение или е вярно, или не е (утре в края на краищата или ще вали, или не).

**Важно** е още сега да се знае, че вероятността, която един субект съпоставя на едно събитие, винаги е свързана с неговите знания относно изпълнението на някакви предварителни условия. Така например, когато преди хвърлянето на зар казваме, че вероятността да се падне числото 3 е  $1/6$ , това означава, че по-скоро казваме “ако са изгълнени (например) условията зарчето да е с идеално еднакви стени и при хвърлянето да няма никакви смущения и др., вероятността да се падне числото 3 е  $1/6$ ”. Ако се променят знанията на субекта относно изпълнените предварителни условия, вероятността, която той ще асоциира със събитието, също ще се промени.

б) Нека сега си представим, че на следващия ден същият **субект** наблюдава дъжд (събитието е осъществено) и го характеризира така: “Навън вали много слаб дъжд”. Това изречение съдържа неопределеност от типа размитост - какво означава “много слаб дъжд”, колко е “много е слаб” дъжд? То отразява степента на истинност, с която субектът характеризира проявата на събитието. Тази степен се представя като число в интервала  $[0,1]$  и за “слаб дъжд” тя може да е например 0.1, за “сilen дъжд” - 0.8, за “много сilen дъжд” - 1 и т.н. Забележете, че от онтологична гледна точка, това допуска както дадено събитие, така и неговото обратно събитие да бъдат едновременно осъществени в някаква степен.

в) **Изречението** “С вероятност 70% утре ще вали сilen дъжд” съдържа несигурност и от случаен, и от размит характер.

Знания, които съдържат несигурност от случаен (вероятностен) или размит тип ще наричаме съответно вероятностни или размити знания, а разсъжденията с такива знания - вероятностни или размити разсъждения. Забележете, че формалните съждения, с които работят вероятностните системи, не са принципно различни от формалните съждения в съждителното смятане - те са булеви променливи. От практическа гледна точка, това ограничение е значително, защото сензорните показания на един агент (например, позиционни координати, данни за скорост, сила, температура, налягане и т.н.) обикновено не са булеви, а непрекъснати. За да може да се приложат системите, основани на правила в такива случаи, е необходимо да се намери начин разсъждението да се извършва върху реални променливи. Най-очевидната идея за решаване на въпроса, е да се въведат допълнителни булеви променливи, които дискретизират всяка реална променлива.

**Пример:** Ако променливата X приема стойности от интервала  $[-10;10]$ , бихме могли да въведем трите булеви променливи  $PX$ ,  $ZX$ ,  $NX$  по следния начин:

$PX=T(\text{истина})$  ако  $X \geq 2$ , иначе  $PX=F(\text{лъжа})$ .

$ZX=T(\text{истина})$  ако  $2 > X > -2$ , иначе  $ZX=F(\text{лъжа})$ .

$NX=T(\text{истина})$  ако  $X \leq -2$ , иначе  $NX=F(\text{лъжа})$ .

По този начин винаги точно една от тези променливи ще има стойност  $T(\text{истина})$ , а другите две ще имат стойност  $F(\text{лъжа})$ . Смисълът на тези три променливи, е както следва:  $PX=“X$  е положителна”,  $ZX=“X$  е около нулата”,  $NX=“X$  е отрицателна”. Нека е дадена и променливата Y, която на свой ред приема стойности от интервала  $[-5,5]$ . Аналогично въвеждаме булевите променливи  $PY$ ,  $ZY$  и  $NY$ :

$PY=T(\text{истина})$  ако  $Y \geq 1$ , иначе  $PY=F(\text{лъжа})$ .

$ZY=T(\text{истина})$  ако  $-1 < Y < 1$ , иначе  $ZY=F(\text{лъжа})$ .

$NY=T(\text{истина})$  ако  $Y \leq -1$ , иначе  $NY=F(\text{лъжа})$ .

Нека сега опитаме да изразим следната зависимост между променливите X и Y: “Колкото положителна е променливата X, толкова по-отрицателна е променливата Y, и обратно”. Това ще рече, че за отрицателни стойности на X, Y приема положителни стойности, и обратно - за положителни стойности на X, Y приема отрицателни стойности; за стойности на X близки до нулата, стойностите на Y са също близки до нула. Тази зависимост може да се изрази чрез следните три правила:

1.  $NX \rightarrow PY$

2.  $ZX \rightarrow ZY$

3.  $PX \rightarrow NY$

Тук веднага обаче възникват два **проблема**:

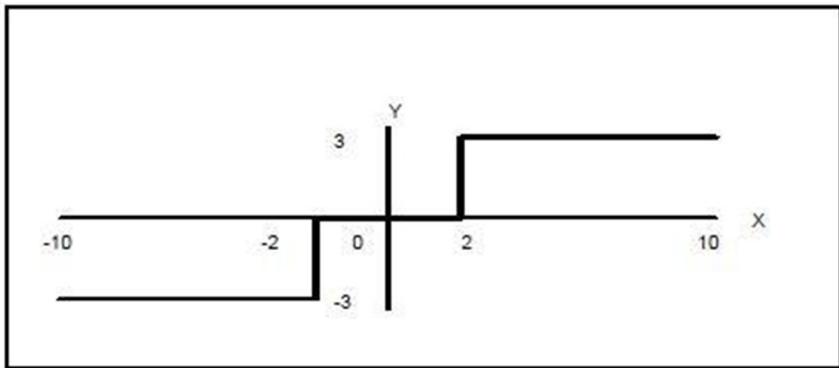
1. **Неточността** на разсъжденията чрез горните три правила. Нека, например, изведем стойността на Y за  $X=5$ : Понеже  $PX=T$ ,  $ZX=F$ ,  $NX=F$ , то по правило 3  $NY=T$ , т.e.  $Y \in [-5, -1]$ . Този резултат не е особено информативен - полученият интервал е твърде голям, за да е полезен, и освен това явно в процеса на разсъждене е изгубена полезна информация. Дискретизацията на реалната променлива X в само три булеви променливи е твърде груба и неточна. Ако не искаме да работим с интервали, можем например да използваме средната стойност за Y, в случая на (-3).

2. **Невъзможността** да се различи стойността на  $X=5$  от  $X=9$ , например, защото едни и същи истинстни стойности ще бъдат присвоени на  $NX$ ,  $ZX$  и  $PX$  и в двата случая ( $F$ ,  $F$ ,  $T$ , съответно). В резултат и в двата случая ще получим интервала  $[1, 5]$  (или неговата средна стойност 3, ако предпочитаме да не работим с интервали).

### Как да постъпим?

Ако увеличим броя на булевите променливи от 3 на 30 например, това наистина ще повиши точността на крайния резултат, но и тък ще доведе до десетократно увеличаване на броя на правилата и съответно забавяне на разсъждението. Освен това, такова решение не би

елиминирило втория важен недостатък на този подход - изходната променлива (консеквентът  $Y$ ) не е гладка функция на входната променлива (антecedента  $X$ ).



Стойността на  $Y$  е средното на съответния интервал. В точките, които служат за разделител на булевите променливи  $PX$ ,  $ZX$  и  $NX$ , изходната променлива  $Y$  изменя стойността си скокообразно (това са точките  $X=-2$  и  $X=2$  в нашия случай). Ако конструираме система за управление, основана на тези правила, тя няма да реагира плавно и постепенно на входните въздействия.

По-скоро, бихме искали да създадем **система**, която винаги преминава плавно от правило към правило в дискретизацията на реалните променливи. Това може да стане, ако можем да охарактеризираме до каква степен са верни антecedентите на правилата, и някак да "пренесем" тези степени на истинност върху консеквентите.

Нека **например**,  $X=9$ . Тази стойност на  $X$  е "много положителна" и "съвсем не нула". С други думи,  $PX$  е "истина до голяма степен", а  $ZX$  е "истина до много малка степен". Съответно бихме очаквали, ако "пренесем" тези степени на истинност през трите правила по-горе,  $Y$  да е "много отрицателна" и "съвсем не нула" - може би около -4.5. Ако пък  $X=3$ , тази стойност е "леко положителна", но и "доста близка до нула". Би трябвало да се очаква, че  $Y$  ще е "леко отрицателна" и "доста близка до нула" - вероятно около -1.5. В този случай  $PX$  е "донякъде истина", но и  $ZX$  "е донякъде истина" също.

За да извършим разсъждение по този начин, е необходимо да дефинираме с езика на математиката какво ще разбираме под фразите "донякъде истина", "истина до голяма степен" и други подобни определения за степен на истинност на булева променлива. Необходимо е също така да намерим начин тези истинностни стойности да се "пренасят" през знака за импликация в логическите правила.

## 11.4 Размити множества и размита логика

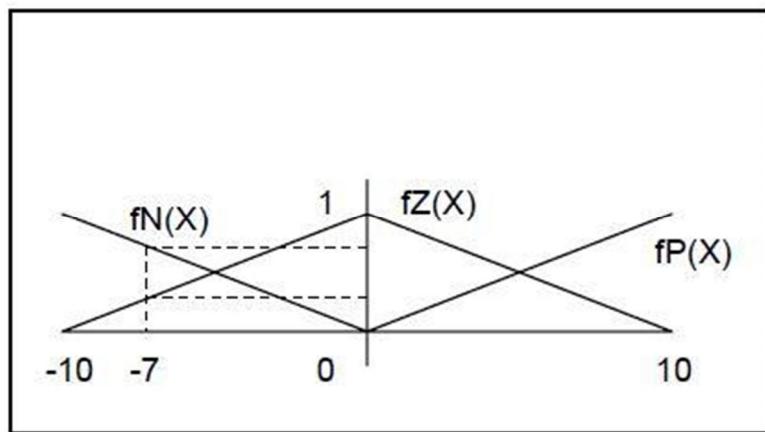
**Размито множество:** двойката  $[S, m(S)]$ , където  $S$  е обикновено множество, а

$m: S \rightarrow [0,1]$  е функция с дефиниционна област  $S$ , която дава резултат в интервала  $[0,1]$ . Функцията  $m(S)$  се нарича характеристична или функция на принадлежност на размитото множество.

**За дискретни** множества  $S$  функцията  $m(S)$  може да се зададе таблично. За непрекъснати множества, например  $X \in S$ ,  $S = [-10, 10]$ , по-удобно е да се зададе аналитична функция, за предпочтение с по-проста форма.

**Пример:** Нека  $S = [-10, 10]$  и  $Q = [-5, 5]$  са две непрекъснати множества и с тях са асоциирани променливите  $X \in S$  и  $Y \in Q$ . „Колкото по-положителна е  $X$ , толкова по-отрицателна е  $Y$  и обратното“. Ще дефинираме следните три размити множества  $PX$ ,  $ZX$ ,  $NX$  за  $X$ .

- $PX = [S, fP(S)]$ ;  $ZX = [S, fZ(S)]$ ;  $NX = [S, fN(S)]$ , където  $S = [-10, 10]$  и
- $fP(X) = 0.1X$  ако  $X > 0$ , иначе 0.
- $fZ(X) = 1 - 0.1X$  ако  $X > 0$ , иначе  $1 + 0.1X$ . (2)
- $fN(X) = -0.1X$  ако  $X < 0$ , иначе 0.



На всяко **размито** множество можем да съпоставим съответна размита променлива. При тази уговорка можем да използваме еднакви имена на променливите и на размитите множества. Нека  $X$  е реална променлива, а  $A_1, A_2, \dots, A_n$  са размити променливи с техните функции на принадлежност  $m_{A_1}(X), m_{A_2}(X), \dots, m_{A_n}(X)$ , съответно. Можем да определим степента на истинност на всяка една от тези променливи за конкретната стойност на  $X$ . Степента на истинност на размита променлива  $A$  за конкретна стойност  $X$  ще означаваме с  $T(A)$ , като  $T(A) = m_A(X)$ .

**Пример:** За  $X = -7$ :  $T(NX) = fN(X) = 0.7$ ,  $T(ZX) = fZ(X) = 0.3$ ,  $T(PX) = fP(X) = 0$ . Аналогично за променливата  $Y$ :  $Y \in Q = [-5, 5]$ , можем да дефинираме следните три размити множества за  $Y$ :

- $PY = [Q, gP(Q)]$ ;  $ZY = [Q, gZ(Q)]$ ;  $NY = [Q, gN(Q)]$ , където  $Q = [-5, 5]$  и
- $gP(Y) = 0.2Y$ , ако  $Y > 0$ , иначе 0
- $gZ(Y) = 1 - 0.2Y$ , ако  $Y > 0$ , иначе  $1 + 0.2Y$  (3)
- $gN(Y) = -0.2Y$ , ако  $Y < 0$ , иначе 0.

## 11.5 Видове изводи в СОЗ



До тук описахме само една част от цялото дърво на изводите в СОЗ. Ето и останалите:

**Дедукция.** Теоретична основа на дедукцията е правилото за извод Modus Ponens (MP). Същност на MP:

(ако A, то B)

$$\frac{A}{B}$$

При това, като модел на твърденията от вида (ако A, то B) обикновено се използва традиционната импликация  $A \rightarrow B$ . В действителност тя не означава непременно причинно-следствена връзка между A и B.

Интерпретаторът на правилата в системите, основани на правила, по същество извършва дедуктивен извод (прав или обратен).

**Абдукция.** Абдукцията е генериране на правдоподобни обяснения за това, което наблюдаваме около нас. Тя може да се разглежда в следната форма:

(ако A, то B)

$$\frac{B}{A}$$

По-точно, абдукцията би трябвало да се разглежда като правило за извод от вида  
(причина  $?x ?y$ )

$$\frac{?y}{?x}$$

**Пример.** Когато хората са пияни, те не могат да пазят равновесие. Ако Джак не може да пази равновесие, бихме могли да предположим, че той е пиян. Естествено, това е само едно предположение, което може да се окаже и невярно (причината за неспособността му да пази равновесие може да бъде съвсем друга).

**Индукция.** Индуктивният извод е опит за обобщение на базата на общи признания, наблюдавани у голям брой конкретни обекти. При натрупване на допълнителни знания за средата достоверността на обобщението може съществено да се повиши. От тази гледна точка може да се твърди, че способността за индуктивен извод е съпоставима със способността на човека за обучение и самообучение.

**Аналогия.** При извода по аналогия на базата на знания за сходство между два обекта по някои признания се генерира хипотезата, че тези обекти са сходни и по други признания, които са установени в единия обект, но все още не са установени в другия. В този смисъл при извода по аналогия се извършва пренасяне (трансформиране) на информация от единния обект към другия.

## 11.6 Представяне на несигурни знания с вероятности

Както споменахме и преди, в работата със несигурни знания не може да бъдем сигурни в истинността на правилото, затова тук трябва да се въвежде мярка за несигурност- вероятността. Това са няколко от методите за представяне на несигурните знания чрез средствата на вероятносите.

- **Случайна променлива.** Това е величина за представяне на знания, която може да има няколко (вкл. безброй много) възможни стойности.
- **Област на променлива:**  $\text{dom}(x)$  = множеството от възможни стойности на  $x$ .
- **Твърдение:** булев израз от присвоявания на променливи ( $x_i = v_i$ ).

Например: (време = дъждовно)  $\vee$  (болест = грип)  $\vee \neg$ (температура = повишена).

- **Вероятност** = мярка за увереност в дадено твърдение (реално число между 0 и 1).  $P(A)=0 \rightarrow 100\%$  увереност, че твърдението  $A$  е лъжа;

$P(A)=1 \rightarrow 100\%$  увереност, че твърдението  $A$  е истина.

- **Вероятностното разпределение** задава вероятността на всяка възможна стойност на променливата. Ако  $\text{dom}(x) = \{v_1, v_2, \dots, v_n\}$ ,

$$\sum_{i=1}^n P(x = v_i) = 1.$$

- **Априорна вероятност** – вероятност при отсъствие на каквато и да е информация.

- **Условна вероятност** – вероятност при наличието на информация за стойностите на други случаини променливи.

Например:  $P(\text{температура} = \text{повишена} \mid \text{болест} = \text{грип})$ . Тоест вероятността за повишена температура при болест грип.

- **Основни зависимости** ( $A, B$  – твърдения):

Ето и някои от основните зависимости във вероятностите:

$$- P(A \vee B) = P(A) + P(B) - P(A \wedge B)$$

Тоест вероятността  $A$  и  $B$  да са едновременно верни е равна на вероятността  $A$  да е вярно + вероятността  $B$  да е вярно – вероятността или  $A$  или  $B$  да са верни.

- ако  $A$  и  $B$  са независими (т.е. знанието на едното не променя вероятността на другото), когато  $P(A \wedge B) = P(A)P(B)$

-  $A$  и  $B$  са несъвместими (т.е. никога не могат да се случат заедно), когато  $P(A \wedge B) = 0$

- Дефиниция на **условна вероятност**:  $P(A \mid B) = P(A \wedge B) / P(B)$

Следователно:  $P(A \wedge B) = P(A \mid B) P(B)$ .

- **Условна независимост** на  $A$  и  $B$  при дадено  $C$ :

ако  $P(A \mid B \wedge C) = P(A \mid C)$  и  $P(B \mid A \wedge C) = P(B \mid C)$

- **Формула (теорема) на Бейс**:  $P(A \mid B) = P(B \mid A)P(A) / P(B)$

- **Вероятностен модел** на предметната област:

- **Атомарно събитие**:  $(x_1 = v_1) \wedge (x_2 = v_2) \wedge \dots \wedge (x_n = v_n)$ , където  $x_i$  са случаини променливи. То описва конкретно състояние на предметната област.

- **Съвместно разпределение**:  $n$ -мерна таблица с  $m_i$  ( $i = 1, 2, \dots, n$ ) с клетки по всяка размерност (ако  $x_i$  има  $m_i$  възможни стойности). Във всяка клетка се записва вероятността на съответното атомарно събитие. Тъй като атомарните събития са несъвместими (т.е. взаимно изключващи се) и таблицата съдържа всички атомарни събития, то сумата от стойностите на всички клетки е 1.

## 11.7 Механизми за извод

- **Използване на съвместното разпределение**

Дадено е съвместното разпределение на няколко случаини променливи, например

	зъбобол = да	зъбобол = не
кариес = да	0.04	0.06
кариес = не	0.01	0.89

Тогава могат да се изчисляват вероятностите на произволни твърдения. Например: (за краткост са пропуснати стойностите на променливите)

$$P(\text{кариес}) = 0.04 + 0.06 = 0.1 \quad (\text{сумата на реда})$$

$$P(\text{кариес} \cap \text{зъбобол}) = 0.04 + 0.06 + 0.01 = 0.11$$

$$P(\text{кариес} | \text{зъбобол}) = P(\text{кариес} \cap \text{зъбобол}) / P(\text{зъбобол}) = 0.04 / (0.04 + 0.01) = 0.8$$

### • Използване на формулата на Бейс

- Дадени са:

$e$  – множество от симптоми ( $e = e_1 \wedge e_2 \wedge \dots \wedge e_k$ ) и  $d_1, d_2, \dots, d_n$  – изчерпващо множество от диагнози. Предполага се, че елементарните симптоми  $\{e_i\}$  са независими. Известни са  $P(d_i)$  и  $P(e|d_i)$  за  $i = 1, \dots, n$  (по-точно,  $P(e_j|d_i)$  за  $j = 1, \dots, k$  и  $i = 1, \dots, n$ ).

- **Задачата** е да се пресметнат  $P(d_i|e)$ ,  $i = 1, \dots, n$  и да се намери най-вероятната диагноза при дадените симптоми  $e$ .

- Според формулата на **Бейс**

$$P(d_i | e) = P(d_i)P(e|d_i) / P(e) \text{ за всяко } i = 1, \dots, n$$

- Предполага се, че елементарните симптоми  $\{e_i\}$  са независими, следователно

$$P(e | d_i) = \prod_{j=1}^k P(e_j | d_i) \text{ за всяко } i = 1, \dots, n$$

-  $P(e)$  може да се намери по следния начин:

$$\sum_{i=1}^n P(d_i | e) = \sum_{i=1}^n \frac{P(d_i)P(e|d_i)}{P(e)} = 1, \text{ следователно}$$

$$P(e) = \sum_{i=1}^n P(d_i)P(e | d_i)$$

### • Пример

вероятност	здрав	грип	алергия
$P(d)$	0.9	0.05	0.05
$P(\text{кихане} d)$	0.1	0.9	0.9
$P(\text{кашлица} d)$	0.1	0.8	0.7
$P(\text{температура} d)$	0.01	0.7	0.4

Нека симптомите ‘ $e$ ’ са кихане и кашлица без повишена температура.

Тогава

$$e = e_1 \wedge e_2 \wedge e_3,$$

$$e_1 = \text{кихане}, e_2 = \text{кашлица}, e_3 = \neg(\text{повишена температура})$$

$$d_1 = \text{здрав}, d_2 = \text{грип}, d_3 = \text{алергия}$$

$$P(\text{здрав} | e) = \frac{P(\text{здрав})P(e | \text{здрав})}{P(e)} = \frac{(0.9)P(e | \text{здрав})}{P(e)},$$

$$P(e | \text{здрав}) = \prod_{j=1}^3 P(e_j | \text{здрав}) = (0.1)(0.1)(1-0.01)$$

Следователно,

$$P(\text{здрав}/e) = \frac{(0.9)(0.1)(0.1)(0.99)}{P(e)} = \frac{0.0089}{P(e)}$$

$$P(\text{грип}/e) = \frac{(0.05)(0.9)(0.8)(0.3)}{P(e)} = \frac{0.01}{P(e)}$$

$$P(\text{алергия}/e) = \frac{(0.05)(0.9)(0.7)(0.6)}{P(e)} = \frac{0.019}{P(e)}$$

$$P(e) = \sum_{i=1}^3 P(d_i)P(e|d_i) = 0.0089 + 0.01 + 0.019 = 0.0379$$

Тоест

$$P(\text{здрав} | e) = 0.23; P(\text{грип} | e) = 0.26; P(\text{алергия} | e) = 0.50$$

- **Проблем:** предположението за независимост на елементарните симптоми, е прекалено силно и нереалистично.

## 11.8 Ресурси

<http://fa.tu-sofia.bg/DPD/courses/AI>

[http://en.wikipedia.org/wiki/Bayes'\\_theorem](http://en.wikipedia.org/wiki/Bayes'_theorem)

<http://www.fmi.uni-sofia.bg/Members/marian>

## **12 Бейсови мрежи.**

Представяне на условните вероятности като структура от данни, максимално удобна за тяхното използване.

### **Съдържание**

#### Съдържание

[Увод в терминологията](#)

[Въведение](#)

[Дефиниция](#)

[Представяне на знания в несигурен домейн](#)

[Видове извод в Бейсови мрежи](#)

[Семантика на бейсовите мрежи](#)

[Представяне на пълното съвместно разпределение](#)

[Метод за конструиране на бейсови мрежи](#)

[Компактност и подредба на възлите](#)

[Условно независими връзки в мрежите на Бейс](#)

[Вериги на Марков](#)

[Ефикасно представяне на условни разпределения](#)

[Бейсови мрежи с непрекъснати променливи](#)

[Приложение на Бейсовите мрежи](#)

[Бейсови мрежи | Речник](#)

[Бейсови мрежи | Ресурси](#)

### **12.1 Увод в терминологията**

В тази тема ще използваме следните понятия:

**Случайна променлива** ще наричаме величина в езика за представяне на знания, която може да има няколко (вкл. безброй много) възможни стойности.

**Област на променлива(домейн)** ще наричаме множеството от възможни стойности на  $x$  и ще бележим с  $\text{dom}(x)$ .

**Твърдение** ще наричаме булев израз от присвоявания на променливи ( $x_i = v_j$ ). Например: (време = дъждовно)  $\vee$  (болест = грип)  $\vee \neg$  (температура = повишена).

**Вероятност** ще наричаме мярка за увереност в дадено твърдение (реално число между 0 и 1).  $P(A)=0$  означава 100% увереност, че твърдението  $A$  е лъжа, а  $P(A)=1$  означава 100% увереност, че твърдението  $A$  е истина.

**Вероятностното разпределение** задава вероятността на всяка възможна стойност на променливата. Ако  $\text{dom}(x) = \{v_1, v_2, \dots, v_n\}$ , то  $\sum_{i=1}^n P(x = v_i) = 1$ .

**Априорна вероятност** ще наричаме вероятност при отсъствие на каквато и да е информация.

**Условна вероятност** ще наричаме вероятност при наличието на информация за стойностите на други случаини променливи. Например:  $P(\text{температура} = \text{повишена} \mid \text{болест} = \text{грип})$ .

**Основни зависимости** ( $A, B$  – твърдения):

$$P(A \vee B) = P(A) + P(B) - P(A \wedge B)$$

$A$  и  $B$  са независими (т.е. знанието на едното не променя вероятността на другото), когато  $P(A \wedge B) = P(A)P(B)$

$A$  и  $B$  са несъвместими (т.е. никога не могат да се случат заедно), когато  $P(A \wedge B) = 0$

$$\text{Дефиниция на условна вероятност: } P(A \mid B) = \frac{P(A \wedge B)}{P(B)}$$

$$\text{Следователно, } P(A \wedge B) = P(A \mid B) P(B)$$

**Условна независимост** на  $A$  и  $B$  при дадено  $C$ : ако  $P(A \mid B \wedge C) = P(A \mid C)$  и  $P(B \mid A \wedge C) = P(B \mid C)$

$$\text{Формула (теорема) на Бейс: } P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}$$

Вероятностен модел на предметната област:

**Атомарно събитие** ще наричаме  $(x_1 = v_1) \wedge (x_2 = v_2) \wedge \dots \wedge (x_n = v_n)$ , където  $x_i$  са случаини променливи. Описва конкретно състояние на предметната област.

**Съвместно разпределение** ще наричаме  $n$ -мерна таблица с  $m_i$  ( $i = 1, 2, \dots, n$ ) клетки по всяка размерност (ако  $x_i$  има  $m_i$  възможни стойности). Във всяка клетка се записва вероятността на съответното атомарно събитие. Тъй като атомарните събития са несъвместими (т.е. взаимно изключващи се) и таблицата съдържа всички атомарни събития, то сумата от стойностите на всички клетки е 1.

## 12.2 Увод

Байсовите мрежи са кръстени на Реверънд Томас Бейс (1702-1761), теолог и математик, живял през 18ти век. Той въвежда правилото на Бейс, което е в основата на Байсовите мрежи. Терминът Байсови мрежи е въведен от Джудеа Пърл през 1985 година, за да набледне на три аспекта:

1. често субективната природа на подаваната информация
2. разчитането на правилото на Бейс като базис за обновяване на информация
3. разликата между случаини и основани на доказателства модели на разсъждения.

Нека си представим света като съвкупност от случайни величини и случайни събития, които се случват с определена вероятност. В него обаче има определено количество зависимости и поредност, тъй като явленията всъщност не са напълно случаен и независими. Тези зависимости заедно с излишеството на информация обуславят реда и предсказуемостта в света. Събитията могат да се класифицират във вероятностни разпределения, по които могат да се правят предположения за други събития и случайни величини от същия домейн. В данни, породени от живи и разумни същества, има изявени зависимости и предсказуемост.

Тази тема въвежда систематичен начин за представяне на независими и условно независими връзки в опростените вероятностни представяния на света експлицитно под формата на бейсови мрежи. Дефинираме синтаксиса и семантиката на мрежите и показваме как те могат да се използват, за да се обхванат несигурните знания по естествен и ефективен начин.

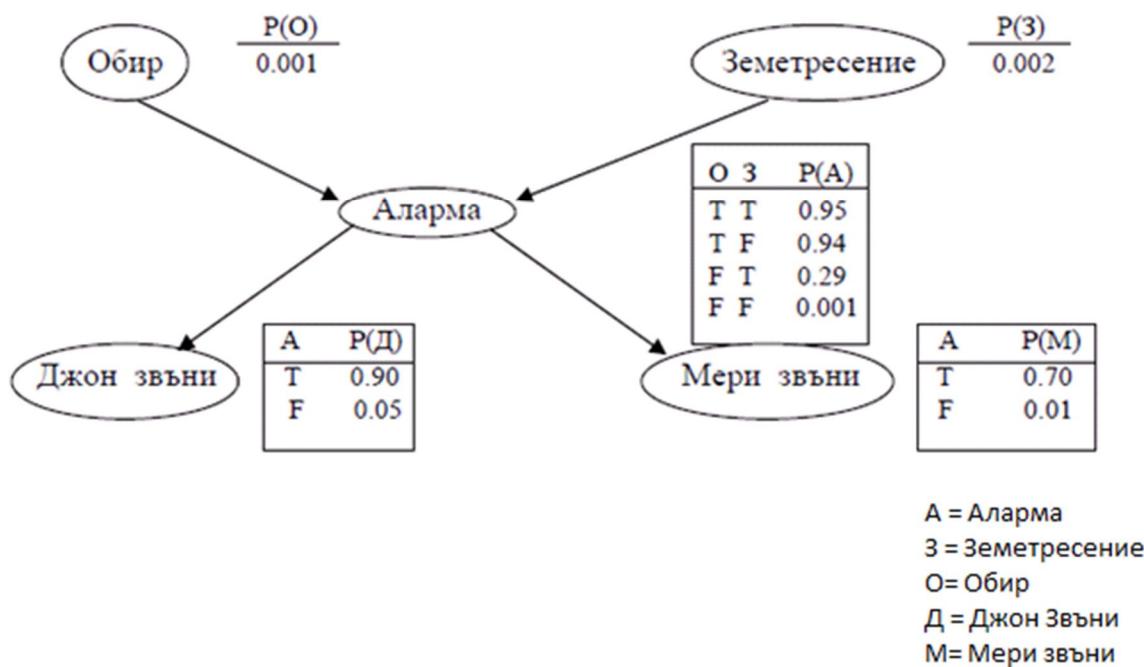
### **12.3 Дефиниция**

Бейсовата (вероятностна) мрежа представлява насочен ацикличен граф, представен от наредената двойка  $(V, E)$ , където  $V$  е множеството от върхове (възли) на графа, а  $E$  е множеството от дъги, съединяващи върховете. Всеки възел на мрежата съдържа име на променлива. Дъгите от мрежата задават причинно-следствени връзки. Бейсовите мрежи се използват за представяне на зависимостите между тези променливи с цел компактно описание на съвместното им разпределение. Интуитивното значение на дъгата от възела  $X$  към възела  $Y$  е, че  $Y$  зависи от стойността на  $X$  или с други думи -  $X$  оказва директно влияние върху  $Y$ . За всеки възел е дефинирана таблица с условно вероятностно разпределение, която показва зависимостта на променливата от всички родители на този възел. Ако променливата няма предшественици, то за нея е дефинирана таблица с априорни вероятности.

### **12.4 Представяне на знания в несигурна област**

Всеки възел от мрежата съдържа таблица на условно вероятностно разпределение, която е дефинирана само върху неговите родители. Така вместо да се съхранява една голяма таблица на съвместно вероятностно разпределение за всички променливи, всеки възел съхранява своя малка таблица на условно вероятностно разпределение, която е дефинирана само върху възлите, от които той е зависим.

Ще разгледаме следния пример: Нека в жилището си имате нова монтирана алармена система. Тя е чувствителна, така че реагира както на опит за проникване в жилището (в частност при обир), така и при земетресения (дори слаби). Нека имате също и двама съседи - Джон и Мери, които са обещали да Ви се обаждат по телефона винаги когато чуят, че алармата във Вашето жилище се е включила. Джон винаги се обажда, когато чуе звука на алармата, но понякога го бърка със звука на телефона и тогава също се обажда. Мери понякога слуша силна музика и тогава е възможно да не чуе, че алармата в дома Ви се е включила. Така ако е известно кой от двамата Ви се е обадил, може да установите вероятността в жилището Ви да е извършен обир.



фигура 1

Нека да разгледаме топологията на мрежата. В примера с алармената система, топологията показва, че обирът и земетресенията влияят директно върху вероятността алармата да се включи, но дали Джон и Мери ще се обадят, зависи само от алармата. Мрежата от фигура 1 представя нашите предположения, че Джон и Мери не забелязват нито кражбите, нито малките земетресения директно. Мрежата няма възли, които да отговарят на това, дали Мери слуша музика в момента или дали телефонното звънене обърква Джон. Тези фактори са обхванати в несигурността, свързана с връзките от *Аларма* към *Мери звъни* и *Джон звъни*. Вероятностите всъщност обобщават потенциалното безкрайно множество от обстоятелства, в които алармата може да не се включи (спиране на тока, срязани кабели и т.н.) или Джон и Мери да не успеят да звъннат (да са на обяд или ваканция и т.н.).

На фигура 1 всяко разпределение е показано под формата на таблица на условната вероятност. Всеки ред в таблицата съдържа условната вероятност на стойността на всеки възел при условен случай. Условният случай е възможна комбинация от стойности на родителските възли. Сумата на всеки ред в таблицата трябва да е 1, защото записите представляват изчерпващо множество от случаи на променливата. За булеви променливи ако приемем, че вероятността за истина е  $p$ , то вероятността за лъжа е  $1-p$ , затова често пропускаме второто число, както във фигура 1. Като цяло таблица от булеви променливи с  $k$  булеви родители съдържа  $2^k$  независими специфични вероятности. Възел без родители има само един ред в таблицата на априорните си вероятности, представящ вероятностите за всяка възможна стойност на променливата.

## 12.5 Видове изводи в Бейсовите мрежи

Съществуват четири вида изводи в Бейсовите мрежи. Това са диагностика, предсказване, между причинен извод и смесен извод. Диагностиката представлява вероятността от следствието към причината. В примера с алармената система това може да бъде вероятността  $P(\text{Обир} | \text{Джон звъни})$ . Предсказването е вероятността от причината към следствието, например  $P(\text{Джон звъни} | \text{Обир})$ . Междупричинният извод е вероятността между причините за дадено следствие. Пример за такъв извод е вероятността  $P(\text{Обир} | \text{Земетресение})$ . Смесеният извод е комбинация на предните три извода, например  $P(\text{Аларма} | \text{Джон звъни} \wedge \neg \text{Земетресение})$ .

## 12.6 Семантика на Бейсовите мрежи

Описахме какво е мрежа, но не и какво означава. Има два начина, по които можем да разберем семантиката на Бейсовите мрежи. Първият е да разгледаме мрежата като представяне на съвместно вероятностно разпределение. Вторият е да се гледа на нея като на колекция от твърдения, описващи условно независими връзки между променливи в нея. Двета начина са еквивалентни, но първият се оказва полезен в разбирането на това как да се построят мрежите, докато вторият е полезен в проектирането на извода на процедурите.

## 12.7 Представяне на пълното съвместно разпределение

Бейсовата мрежа осигурява пълно описание на домейна. Всеки ред в таблицата на пълното съвместно разпределение може да бъде изчислен чрез информациите в мрежата. Нека  $x_1, x_2, \dots, x_n$  са случаини променливи и  $P(v_1, v_2, \dots, v_n)$  е съвместната вероятност те да получат съответно стойности  $v_1, v_2, \dots, v_n$ . Тогава

$$P(v_1, v_2, \dots, v_n) = \prod_{i=1}^n P(v_i | \text{Parents}(x_i)) \quad (1)$$

където  $P(v_i | \text{Parents}(x_i))$  е условната вероятност за  $x_i = v_i$  при условие, че са дадени стойностите на родителските променливи  $\text{Parents}(x_i)$  на  $x_i$ . Всеки множител на пълното съвместното разпределение е представен като умножение на подходящи елементи от таблицата на условната вероятност в Бейсовата мрежа. Таблицата на условната вероятност осигурява декомпозирано представяне на съвместното разпределение.

За да илюстрираме това, можем да пресметнем вероятността алармата да се включи, но да няма нито обир, нито земетресение, и Джон и Мери да се обадят:

$$P(\text{Джон звъни} \wedge \text{Мери звъни} \wedge \text{Аларма} \wedge \neg \text{Обир} \wedge \neg \text{Земетресение}) = \\ P(\text{Джон звъни} | \text{A}).P(\text{Мери звъни} | \text{A}).P(\text{Аларма} | \text{A}).P(\neg \text{Обир} | \text{A}).P(\neg \text{Земетресение} | \text{A}) = \\ P(\text{Джон звъни} | \text{A}).P(\text{Мери звъни} | \text{A}).P(\text{Аларма} | \text{A}).P(\neg \text{Обир} | \text{A}).P(\neg \text{Земетресение} | \text{A}) = \\ (0.9)(0.7)(0.001)(0.999)(0.998) = 0.000628.$$

## 12.8 Метод за конструиране на бейсови мрежи

Формула (1) обяснява какво означава Бейсова мрежа, но не обяснява как да се построи Бейсова мрежа по такъв начин, че резултатното съвместно разпределение да е добро представяне на дадения домейн. Ще покажем, че уравнение (1) включва определени условно-

независими връзки, които могат да бъдат използвани при конструиране топологията на мрежата. Ще използваме правилото за произведение, което гласи, че за твърденията  $a$  и  $b$  е в сила:

$$P(a \wedge b) = P(a | b)P(b) \text{ при } P(b) > 0,$$

за да представим условното разпределение по следния начин:

$$P(x_1, \dots, x_n) = P(x_n | x_{n-1}, \dots, x_1)P(x_{n-1}, \dots, x_1)$$

Повтаряме процеса като редуцираме всяка конюнкция до произведение на условна вероятност и конюнкция на по-малко на брой аргументи:

$$P(x_1, \dots, x_n) = P(x_n | x_{n-1}, \dots, x_1)P(x_{n-1} | x_{n-2}, \dots, x_1) \dots P(x_2 | x_1)P(x_1) = \prod_{i=1}^n P(x_i | x_{i-1}, \dots, x_1)$$

Формулата е истина за всяко множество от произволни променливи и се нарича правило на веригата. От формула (1) виждаме, че спецификацията на съвместното разпределение е еквивалентна на общото твърдение, че за всяка променлива  $X_i$  в мрежата, е изпълнено

$$P(X_i | X_{i-1}, \dots, X_1) = P(X_i, \text{Parents}(X_i)), \quad (2)$$

където  $\text{Parents}(X_i) \subseteq \{X_{i-1}, \dots, X_1\}$ .

Формула (2) показва, че Бейсовата мрежа е коректна репрезентация на домейна, само ако всеки възел е условно независим от неговите предшественици при съответната подредба на възлите в мрежата. Следователно, за да построим Бейсова мрежа с коректната структура на домейна, трябва да изберем родители за всеки възел, такива че да е изпълнено горепосоченото свойство. Родителите на възела  $X_i$  трябва да съдържат всички тези възли в  $X_1, \dots, X_{i-1}$ , които директно влияят на  $X_i$ . Например да предположим, че сме завършили мрежата на фигура 1, с изключение на избора на родители за *Мери звъни*. *Мери звъни* определено е повлияна от това дали има земетресение или обир, но не е директно повлияна. Нашето знание за домейна ни подсказва, че тези събития повлияват на обаждането на Мери единствено чрез ефекта на алармата. Също така, ако вземем предвид състоянието на алармата, дали Джон ще се обади не повлиява на позвъняването на Мери. Твърдим, че следващото твърдение, изразяващо условна независимост, е в сила:

$$P(\text{Мери звъни} | \text{Джон звъни}, \text{Аларма}, \text{Земетресение}, \text{Обир}) = P(\text{Мери звъни} | \text{Аларма})$$

При дадени стойности, срещани в подмножество от променливите, можем да определим вероятността на стойностите на друго подмножество от променливите. Например можем да определим: каква е вероятността при обир Джон да се обади; вероятността Джон да се обади и да има обир; да има обир и да има земетресение; вероятността алармата да се включи и Джон да звънне или да няма земетресение и други.

## 12.9 Компактност и подредба на възлите

Мрежата на Бейс често може да бъде далеч по-компактна от колкото пълното съвместно разпределение. Това свойство я прави предпочита на за справяне с домейни с много променливи.

При Бейсовите мрежи, можем да твърдим, че в повечето домейни всяка случайна променлива е директно повлияна от най-много  $k$  други, за някоя константа  $k$ . Ако предположим, че имаме

и  $n$  булеви променливи, то тогава количеството информация, което е нужно, за да се определи всяка таблица на условна вероятност, ще се състои най-много от  $2^k$  стойности и пълната мрежа може да бъде определена от  $n*2^k$  стойности. Съвместното разпределение от своя страна се представя чрез  $2^n$  стойности. Нека предположим, че имаме  $n=30$  възела, всеки с по пет родители ( $k=5$ ). Тогава мрежата на Бейс изисква 960 числа, докато пълното съвместно разпределение изисква повече от милиард. Ето защо Бейсовите мрежи често са предпочитан подход за представяне на домейни с много променливи.

Има домейни, в които всяка променлива може да бъде повлияна директно от всички други, така че мрежата да е изцяло свързана. Тогава определянето на таблиците на условните вероятности ще изисква същото количество информация като определянето на пълното съвместно разпределение. В някои домейни може да има слаби зависимости, които трябва да бъдат включени чрез добавянето на нова връзка. Но ако тези зависимости са много незначителни, тогава не си струва да добавяме допълнително усложнение в мрежата за достигане на по-голяма прецизност.

Например някои може да оспорят нашата мрежа за алармена система, защото ако има земетресение, Джон и Мери не биха се обадили дори да чуят алармата, защото ще предположат, че земетресението е причината. Дали да се добави връзка от *Земетресение* към *Джон звъни* и *Мери звъни*(и така да увеличим таблиците) зависи от важността да вземем по-прецизни вероятности с цената на добавяне на допълнителна информация.

Дори в локално структуриран домейн, изграждането на локално структурирана Бейсова мрежа не е тривиална задача. Ние изискваме не само всяка променлива да бъде директно повлияна само от малко други, но също така и топологията на мрежата наистина да отразява тези директни влияния с подходящо множество от родители. Заради начина, по който работи процедурата за построяване на Бейсови мрежи, „директните повлиятори“ ще трябва да бъдат добавени първи към мрежата, ако ще стават родители на възела, на който влияят.

Затова правилният ред, по който да се добавят възли е да се добавят първо корените, след това променливите, на които те влияят и така нататък, докато достигнем до листата, които не оказват директно влияние върху други променливи.

Какво се случва, ако изберем погрешна подредба? Нека разгледаме примера с алармената система отново. Да предположим, че решим да добавим възлите в реда *Мери звъни*, *Джон звъни*, *Аларма*, *Обир*, *Земетресение*. Тогава получаваме до известна степен по-сложна мрежа, изобразена на фигура 2.1. Процесът се изпълнява както следва:

Добавяме *Мери звъни*, която няма родители.

Добавяме *Джон звъни*. Ако Мери звънне, това вероятно означава, че алармата се е включила, което разбира се ще направи по-вероятно Джон да звънне. Ето защо *Джон звъни* има нужда от *Мери звъни* като свой родител.

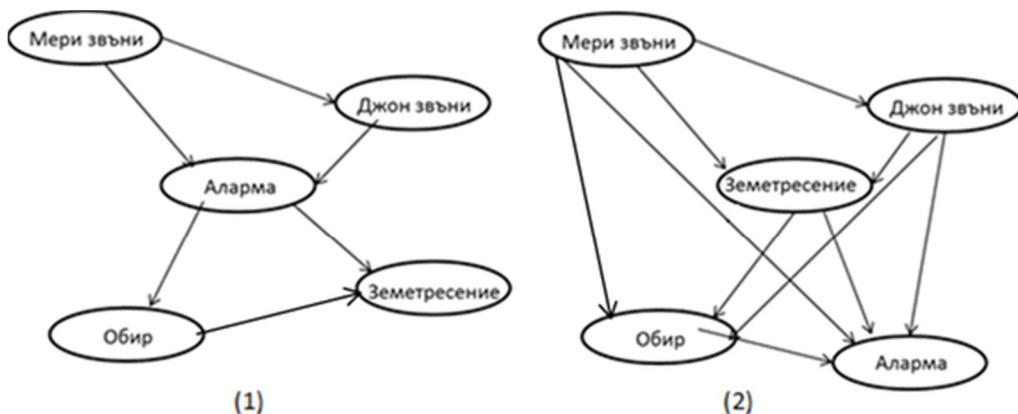
Добавяме *Аларма*. Очевидно, ако двамата звъннат, е по-вероятно алармата да се е включила, отколкото ако само единият или нито единият от двамата звънне, така че имаме нужда и от двамата, *Мери звъни* и *Джон звъни*, като родители.

Добавяме *Обир*. Ако знаем състоянието на алармата, тогава позвъняването от Джон или Мери може да ни даде информация за това дали телефонът звъни или Мери слуша музика, но не и за обира.

$$P(\text{Обир} | \text{Аларма}, \text{Джон звъни}, \text{Мери звъни}) = P(\text{Обир} | \text{Аларма})$$

Следователно имаме нужда само от *Аларма* като родител.

Добавяме *Земетресение*. Ако алармата се е включила, то е по-вероятно да е имало земетресение. Но ако знаем, че е имало грабеж, това би обяснило защо алармата се е включила, и вероятността да е имало земетресение би била малко под нормалната. Следователно имаме нужда от *Аларма* и *Обир* като родители.



Фигура 2

Крайният резултат на мрежата има две връзки повече от оригиналната на фигура 1 и изисква още три вероятности, които трябва да се определят. По-лошото е, че някои от дългите представляват незначителни връзки, които изискват трудни и неестествени вероятностни решения, като оценяване на вероятността на *Земетресение* спрямо *Обир* и *Аларма*.

Вторият случай на фигура 2 показва много лошо подреждане: *Мери звъни*, *Джон звъни*, *Земетресение*, *Обир*, *Аларма*. Мрежата изисква 31 различни вероятности, които да бъдат определени – точно толкова колкото в пълното съвместно разпределение. Важно е да се разбере, обаче, че всяка една от трите мрежи може да представи абсолютно същото съвместно разпределение. Последните две версии се провалят в представянето на всички условно независими връзки и затова в крайна сметка определят прекалено много ненужни стойности.

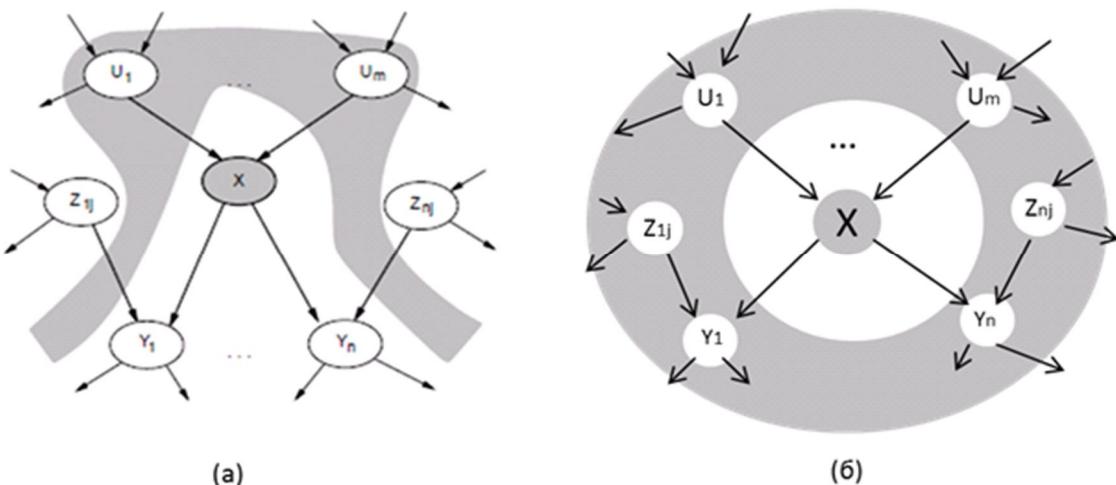
## 12.10 Условно независими връзки в мрежите на Бейс

Представихме „числовата“ семантика на Бейсовите мрежи по отношение на представянето на пълно съвместно разпределение, както във формула(1). Използвайки тази семантика за получаване на метод за конструиране на Бейсови мрежи, сме доведени до заключението, че един възел е условно независим от неговите предшественици спрямо неговите родители. Оказва се, че можем да отидем и в другата посока. Можем да започнем от „топологична“ семантика, която определя условно независимите връзки, кодирани от структура на граф, и от

тях можем да получим „числовата“ семантика. Топологичната семантика се задава чрез следните две спецификации, които са еквивалентни:

1. Един възел е условно независим от възлите, които не са негови наследници, спрямо неговите родители. Например възелът *Джон звъни* е независим от възлите *Обир* и *Земетресение* спрямо стойността на *Аларма*. Това се нарича още глобална семантика.
2. Един възел е условно независим от всички други възли в мрежата спрямо неговите родители, деца и родителите на децата, т.е. спрямо одеалото на Марков. Например *Обир* е независим от *Джон звъни* и *Мери звъни* спрямо *Аларма* и *Земетресение*. Това се нарича още локална семантика.

Тези спецификации са илюстрирани на фигура 3.



Фигура 3

На фигура 3 (a) възелът  $X$  е условно независим от възлите, които не са неговите наследници (например възлите  $Z_{ij}$ ), спрямо неговите родители ( $U_i$  в сивата област). На фигура 3(б) възелът  $X$  е условно независим от всички други възли в мрежата спрямо неговото одеало на Марков (сивата област).

## 12.11 Вериги на Марков

Нека имаме претеглен насочен граф без цикли. Нека теглата са вероятности за преход, а сборът от коефициентите на ребрата, излизящи от всеки възел, е 1. Нека редицата  $X_1, X_2, X_3$  от случаини величини има свойството на Марков.  $X_1, X_2, X_3$  са конкретни дискретни състояния, а  $x_1, x_2, x_3$  са вероятности да се премине в това състояние в дадена стъпка. Тогава:

**Свойство на Марков** наричаме вероятността за преминаване в състояние  $x_{n+1}$ . Тя зависи единствено от това кое е предходното състояние:

$$\Pr(X_{n+1}=x | X_1=x_1, X_2=x_2, \dots, X_n=x_n) = \Pr(X_{n+1}=x | X_n=x_n)$$

**Вериги на Марков** наричаме описание на процеси, които се променят в дискретно зададено време, на тактове.

**Вериги на Марков от  $m$ -ти ред** наричаме новото състояние, което зависи само от не повече от  $m$  предходни състояния.

**„Одеало“ на Марков** наричаме възел в мрежа на Бейс, неговите родители, наследници и другите родители на наследниците му.

## 12.12 Ефективно представяне на условни разпределения

Дори ако максималният брой на родителите  $k$  е малък, запълването на таблицата на съвместното вероятностно разпределение за всеки възел ще изиска до  $O(2^k)$  числа. Въщност, това е най-лошият възможен сценарий, в който връзката между родителите и детето е напълно случаена. Обикновено такива връзки се описват от канонично разпределение, което подхожда на съответния шаблон. В такива случаи цялата таблица може да бъде определена като се именова шаблона и се осигурят няколко параметъра – много по-лесно от осигуряването на експоненциален брой параметри.

Най-простият пример за ефективно представяне на условни разпределения е свързан с детерминистичните възли. Детерминистичният възел има стойност, точно определена от стойностите на родителите му, без несигурности. Връзката може да бъде логическа: например връзката между родителските възли канадец, американец и мексиканец и възела на детето – североамериканец е просто, че детето е дизюнкция на родителите си. Връзката също може да бъде числовая: например, ако родителските възли са цените на определен модел автомобил на няколко продавачи, а детето е цената, която купувачът плаща, тогава възелът на детето е минимумът от стойностите на родителите.

Несигурните връзки често могат да бъдат определени като „шумни“ логически връзки. Стандартният пример е с „шумна-или“ връзка, която е обобщение на логическото ИЛИ. В предикатната логика, бихме могли да кажем, че Температура е истина, тогава и само тогава, когато Настинка, Грип или Малария е истина. Моделът на „шумно-или“ позволява несигурност относно възможността на всеки родител да позволи на детето да бъде истина – причинно-следствената връзка между родител и дете може да бъде потисната, и така например един пациент може да има Настинка, но да няма Температура. Моделът прави две предположения. Първото предполага, че всички възможни причини са изброени. Второто предполага, че потискането на всеки родител е независимо от потискането на другите родители: например това, което потиска Малария от причиняването на Температура, е независимо от това, което потиска Настинка от причиняването на Температура. Спрямо тези предположения, Температура е лъжа тогава и само тогава, когато всичките й родители са потиснати, а вероятността за това е произведение от вероятностите за потиснатост на всеки един родител. Нека да предположим, че тези вероятности на тези индивидуални потиснатости са както следва:

$$P(\neg \text{Температура} | \text{Настинка}, \neg \text{Грип}, \neg \text{Малария}) = 0.6,$$

$$P(\neg \text{Температура} | \neg \text{Настинка}, \text{Грип}, \neg \text{Малария}) = 0.2,$$

$$P(\neg \text{Температура} | \neg \text{Настинка}, \neg \text{Грип}, \text{Малария}) = 0.1,$$

Тогава от тази информация и „шумно-или“ предположенията, цялата таблица на условната вероятност може да бъде построена. Таблица 1 показва как:

Настинка	Грип	Малария	$P(\text{Температура})$	$P(\neg \text{Температура})$
Лъжа	Лъжа	Лъжа	0.0	1.0
Лъжа	Лъжа	Истина	0.9	0.1
Лъжа	Истина	Лъжа	0.8	0.2
Лъжа	Истина	Истина	0.98	$0.02 = 0.2 * 0.1$
Истина	Лъжа	Лъжа	0.4	0.6
Истина	Лъжа	Истина	0.94	$0.06 = 0.6 * 0.1$
Истина	Истина	Лъжа	0.88	$0.12 = 0.6 * 0.2$
Истина	Истина	Истина	0.988	$0.012 = 0.6 * 0.2 * 0.1$

таблица 1

Като цяло, „шумните“ логически връзки, в които променлива зависи от  $k$  на брой родители, може да бъде описана, използвайки  $O(k)$  параметъра, вместо  $O(2^k)$  за пълната таблица на условната вероятност.

### 12.12.1     Бейсови мрежи с непрекъснати променливи

Много реални проблеми включват непрекъснати количества, като например височина, маса, температура. По дефиниция, непрекъснатите променливи имат безкраен брой възможни стойности, така че е невъзможно да се определят условните вероятности за всяка стойност експлицитно. Един възможен начин за справяне с непрекъснатите променливи е те да се избягват чрез използване на дискретизация – това е разделянето на възможните стойности във фиксирано множество от интервали. Например температурата може да бъде разделена на по-малко от 0 градуса, между 0 и 100 градуса и над 100 градуса. Дискретизацията понякога е адекватно решение, но често последват значителни загуби в точността и големи таблици на условни вероятности. Другото решение е да се определят стандартни фамилии от вероятностни функции на плътност, които са определени от краен брой параметри.

Мрежа с дискретни и непрекъснати променливи се нарича хибридна бейсова мрежа. За да определим хибридна мрежа, трябва да определим два нови типа разпределения: условно разпределение за непрекъсна променлива спрямо нейните дискретни или непрекъснати родители; и условно разпределение за дискретна променлива спрямо непрекъснати родители.



фигура 4 Мрежа с дискретни(Субсидия и Продажби) и непрекъснати(Реколта и Цена) променливи

Нека разгледаме следния пример (фигура 4): Нека клиентът си купува плодове, в зависимост от цената им, която зависи от размера на реколтата и от субсидиите. Променливата Цена е непрекъсната и има непрекъснати и дискретни родители. Променливата Продажби е дискретна и има непрекъснат родител.

За променливата Цена е нужно да определим вероятността  $P(\text{Цена} | \text{Реколта}, \text{Субсидия})$ . За да обработим Реколта, трябва да определим как разпределението над цената съзиси от непрекъснатата стойност на  $b$  на Реколта. С други думи, трябва да определим параметрите на разпределението на цената като функция на  $b$ . Най-често срещаният избор за това е чрез линейното Гаусово разпределение. Така чрез него можем да пресметнем вероятността за попадане на случайна променлива в определен интервал от стойности.

## 12.13 Приложение на Бейсовите мрежи

Бейсовите мрежи се прилагат в някои дисциплини като програмирането, медицината, биоинформатиката, обработката на снимки, информационни системи за подпомагане на решения, алгоритми за търсене и дори в управление на бизнеса. Бейсовите мрежи помагат в улесняване на по-доброто разбиране на много сложни събития и феномени.

За пример можем да дадем медицинските проучвания, които използват Бейсови мрежи, за да изобразят връзката на определена болест с няколко симптома. Друг пример е свързан с финансовите аналитици, мениджъри и икономисти, при които Бейсовите мрежи са наложителни фактори в техните процеси за взимане на решения.

Бейсовите мрежи се използват в софтуерни програми като WinBUGS, OpenBUGS, Hugin, Netica и други. WinBUGS е статистически софтуер, използван за Бейсов анализ. Той се базира на проекта BUGS (Bayesian inference Using Gibbs Sampling), който започва през 1989 година. OpenBUGS е open source варианта на WinBUGS. Hugin и Netica също се базира на Бейсови мрежи. Те се използват за подпомагането на решения, медицинска диагностика, анализ на риска и други.

## 12.14 Речник

domain	област на променлива
updating algorithms	алгоритми за обновяване
collection	колекция
statement	твърдение
product rule	правило на произведението
noisy-OR	шумно “ИЛИ”
conditioning case	условен случай
propositional logic	предикатна логика
discretization	дискретизация

## 12.15 Ресурси

Artificial Intelligence: A Modern Approach (2nd Edition) Stuart Russell, Peter Norvig, 2002

Изкуствен интелект, М. Нипева, Д. Шипков, изд. „Интеграл“ Добрич, 1995

<http://www.math.bas.bg/index.html/>

[http://research.twenkid.com/agi\\_2011/Machine\\_Learning\\_basics\\_4-Feb-2011-MTR.pdf](http://research.twenkid.com/agi_2011/Machine_Learning_basics_4-Feb-2011-MTR.pdf)

[http://plaza.patso.org/zed/IIES/mat/books/2k6-7/AI\\_Lecture8.pdf](http://plaza.patso.org/zed/IIES/mat/books/2k6-7/AI_Lecture8.pdf)

[http://fa.tu-sofia.bg/DPD/courses/SIP/SIP\\_L09\\_07.pdf](http://fa.tu-sofia.bg/DPD/courses/SIP/SIP_L09_07.pdf)

[http://en.wikipedia.org/wiki/Bayesian\\_network](http://en.wikipedia.org/wiki/Bayesian_network)

[http://research.twenkid.com/agi\\_2011/Machine\\_Learning\\_basics\\_4-Feb-2011-MTR.pdf](http://research.twenkid.com/agi_2011/Machine_Learning_basics_4-Feb-2011-MTR.pdf)

<http://www.helium.com/items/2109944-what-is-a-bayesian-network>

[http://en.wikipedia.org/wiki/Bayesian\\_network](http://en.wikipedia.org/wiki/Bayesian_network)

<http://en.wikipedia.org/wiki/OpenBUGS>

<http://en.wikipedia.org/wiki/WinBUGS>

## **13 Учене на дърво на решенията. Ансамболово учене.**

Разглеждат се алгоритми, даващи възможност на един агент сам да подобрява бъдещото си представяне на база на събраниите от него знания за света.

### **Съдържание**

[Съдържание](#)

[Учене на дърво на решенията](#)

[Представяне на дървото на решения](#)

[Изразителност на дървото на решения](#)

[Строене на дърво на решения от примери](#)

[Производителност на алгоритъма за строене](#)

[Избиране на „най – добрия“ атрибут](#)

[Пренагаждане](#)

[Ансамболово учене](#)

[Учене на дърво на решенията. Ансамболово учене. | Речник](#)

[Учене на дърво на решенията. Ансамболово учене. | Източници](#)

### **13.1 Учене на дърво на решенията.**

#### **Представяне на дървото на решения**

*Дърво на решения представлява функция, която приема за вход вектор от атрибути и връща „решение“ – една единствена стойност.*

$$F: (A_1, A_2, \dots, A_N) \rightarrow D$$

Попринцип стойностите на атрибутите на входа и изхода на функцията могат да принадлежат както на крайни така и на безкрайни множества. За сега ще разгледаме случая, в който  $A_1, A_2, \dots, A_N$  са от крайно множество, а  $D$  приема точно 2 стойности – **True**, **False** (Булево дърво на решения).

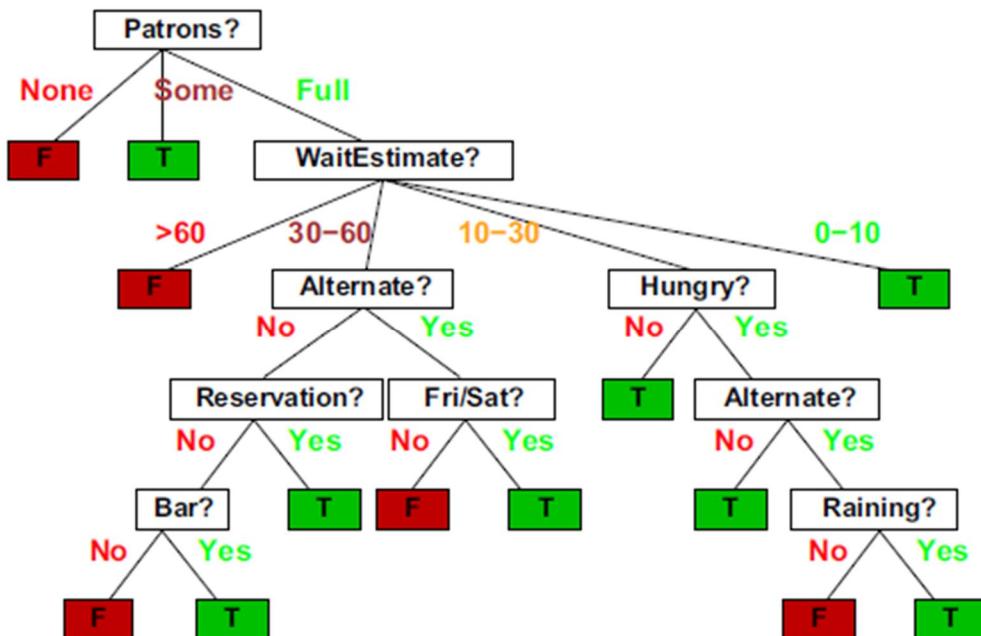
Дървото на решения достига своята цел като изпълнява серия от тестове. Всеки вътрешен възел на дървото отговаря на тест върху някой от входните атрибути  $A_i$  и всяко разклонение от този възел отговаря на някоя стойност от множеството, в което се изменя  $A_i$ .

**Пример:**

За да разясним по – добре представянето на дърво на решенията ще построим такова за предикатната цел „WillWait“. Тази цел ще дава информация дали да чакаме в ресторант при дадени ни определени условия (входни данни). За да построим дървото първо трябва да изброим входните атрибути:

- **A1 Alternate?** – дали имаме алтернативен ресторант наблизо. Стойностите са от множеството {True, False}.
- **A2 Bar?** – дали ресторантът има удобен бар, където да чакаме. Стойностите са от множеството {True, False}.
- **A3 Fri/Sat?** – дали е Петък или Събота. Стойностите са от множеството {True, False}.
- **A4 Hungry?** – дали сме гладни. Стойностите са от множеството {True, False}.
- **A5 Patrons?** – Колко е пълен ресторантът. Стойностите са от множеството {None, Some, Full}.
- **A6 Price?** – ценовия клас на ресторанта. Стойностите са от множеството {\$, \$\$, \$\$\$}.
- **A7 Raining?** – дали навън вали. Стойностите са от множеството {True, False}.
- **A8 Reservation?** – дали имаме резервация. Стойностите са от множеството {True, False}.
- **A9 Type?** – типът на ресторанта. Стойностите са от множеството {French, Italian, Thai, Burger}.
- **A10 WaitEstimate?** – оценка на времето, което ще трябва да чакаме. Стойностите са от множеството {0-10, 10-30, 30-60, > 60}.

Важно е да се отбележи, че всички входни атрибути са с малки допустими множества, което и ще ни позволи да покажем как би изглеждало примерно цяло дърво на решението за този проблем (фиг. 1).



(Фиг. 1)

### Изразителност на дървото на решения

Булевото дърво на решението е логически еквивалентно на допускането, че целта е истина тогава и само тогава, когато входните данни удовлетворяват някой от пътищата водещи от корена до листо със стойност истина. В съждителната логика това изглежда:

$$\text{Goal} = (\text{Path1} \vee \text{Path2} \vee \dots \vee \text{PathK})$$

Тук **Pathi** е конюнкция от тестовете на входните данни, които трябва да се изпълнят, за да се следва пътя. Например най – десния клон на дървото от фиг. 1 може да се запише като:

$$\text{Path} = (\text{Patrons?} = \text{'Full'} \& \text{WaitEstimate} = \text{'0-10'})$$

Това означава, че целия предикатен израз за **Goal** е в дизюнктивна нормална форма, което от своя страна значи, че всяка функция в съждителната логика може да се изрази като дърво на решението.

Дървото на решението дава добър и точен резултат за много разновидности проблеми. За жалост има и функции, които не могат да се представят добре. Например *мажоритарната функция*, която връща истина тогава и само тогава когато повече от половината входове са истина, изисква експоненциално голямо дърво на решението.

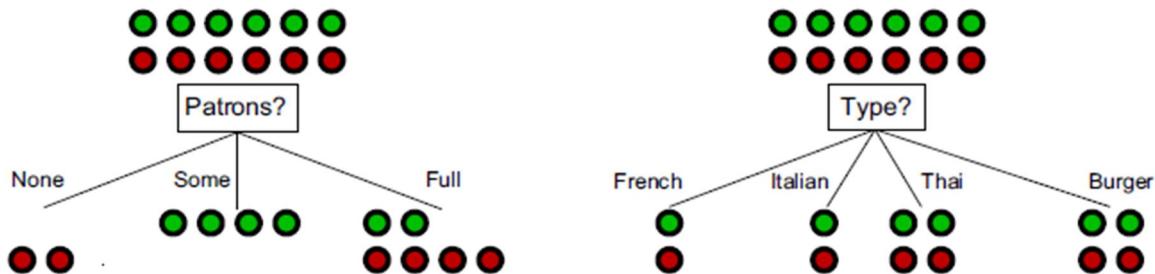
### Строене на дърво на решения от примери

Пример за Булево дърво на решения е двойка  $(\mathbf{X}, \mathbf{Y})$ , където  $\mathbf{X}$  е вектор от стойности за входните атрибути, а  $\mathbf{Y}$  е True или False. Множество от примери за целта „WillWait“ е дадено във фиг. 2.

Example	Attributes										Target WillWait
	Alt	Bar	Fri	Hun	Pat	Price	Rain	Res	Type	Est	
$X_1$	T	F	F	T	Some	\$\$\$	F	T	French	0–10	T
$X_2$	T	F	F	T	Full	\$	F	F	Thai	30–60	F
$X_3$	F	T	F	F	Some	\$	F	F	Burger	0–10	T
$X_4$	T	F	T	T	Full	\$	T	F	Thai	10–30	T
$X_5$	T	F	T	F	Full	\$\$\$	F	T	French	>60	F
$X_6$	F	T	F	T	Some	\$\$	T	T	Italian	0–10	T
$X_7$	F	T	F	F	None	\$	T	F	Burger	0–10	F
$X_8$	F	F	F	T	Some	\$\$	T	T	Thai	0–10	T
$X_9$	F	T	T	F	Full	\$	T	F	Burger	>60	F
$X_{10}$	T	T	T	T	Full	\$\$\$	F	T	Italian	10–30	F
$X_{11}$	F	F	F	F	None	\$	F	F	Thai	0–10	F
$X_{12}$	T	T	T	T	Full	\$	F	F	Burger	30–60	T

(фиг. 2)

Нашата задача е да построим дърво, което спазва примерите и е възможно най – малко (откриването на най – малкото е нерешен проблем заради големия брой на дърветата, които могат да се построят). **Decision-Tree-Learning** е лаком алгоритъм, използващ стратегията разделяй и владей – винаги първо се тества „най – важния“ атрибут като този тест разделя задачата на по – малки подзадачи. Под „най – важен“ атрибут се разбира такъв, който разделя примерите на подмножества с преобладаващ брой True или False резултати. Например на фиг.3 е показано че атрибутът **Type?** не е важен, защото ни оставя с 4 подмножества, за всяко от които трябва да се правят още тестове. От друга страна атрибутът **Patrons?** ни оставя с 3 множества – 2 чиито резултат е ясен (съдържат само положителни/отрицателни тестове) и 1, чието множество е смесено и ще трябва да се правят още тестове. Избирайки по този начин важните атрибути си гарантираме, че крайното дърво ще има минимален брой разклонения.



(Фиг. 3)

Алгоритъма **Decision-Tree-Learning** (показан на Фиг. 4) има 4 отделни случая, на които трябва да се обърне внимание, за да заработи рекурсията:

- Ако всички оставащи примери са положителни/отрицателни тогава сме свършили и можем да отговорим True/False.
- Ако имаме останали няколко положителни и няколко отрицателни примера тогава трябва да изберем „най-добрия“ и да ги разделим по него.
- Ако нямаме останали примери, това означава, че няма пример за тази комбинация от атрибути. В този случай връщаме default-на стойност, която се изчислява на базата на преобладаващия изход (*Модата*) от всички примери използвани за строене на възела родител на текущия. Тези стойности се предават чрез *default* параметъра.
- Ако нямаме останали атрибути, а в множеството от стойности имаме и положителни/отрицателни примери това означава, че тези примери имат същото описание, но различни резултати. Причина за това може да е евентуален шум при доставянето на данните или незнанието ни за определен атрибут, който би разграничили примерите. В този случай най – доброто, което можем да направим е да върнем преобладаващия изход (*Модата*) от оставащите примери.

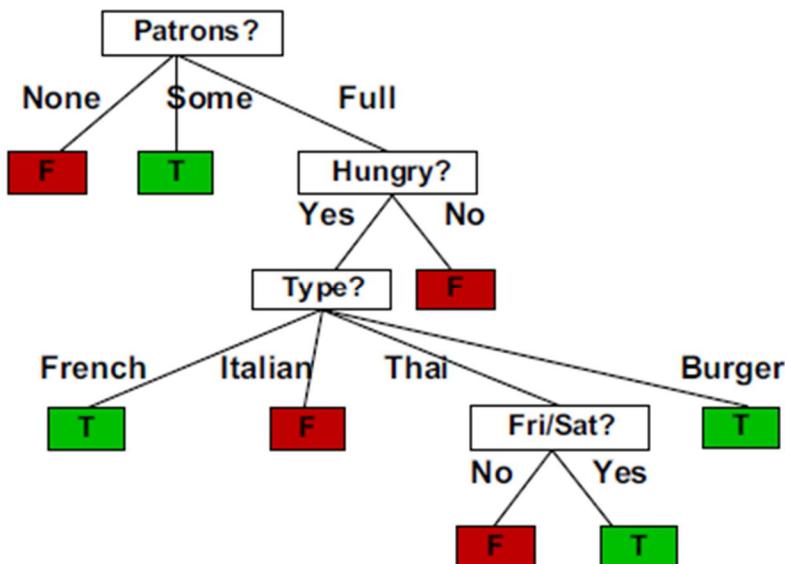
```

function DTL(examples, attributes, default) returns a DT
  if examples is empty then return default
  else if all examples have same class then return class
  else if attributes is empty then return Mode(examples)
  else
    best  $\leftarrow$  Choose-Attribute(attributes, examples)
    tree  $\leftarrow$  a new decision tree with root test best
    for each value vi of best do
      exi  $\leftarrow$  {elements of examples with best = vi}
      subtree  $\leftarrow$  DTL(exi, attributes – best, Mode(examples))
      add a branch to tree with label vi and subtree subtree
  return tree

```

(Фиг. 4)

Така определения **Decision-Tree-Learning** алгоритъм ще построи дървото от фиг. 5 за целта „WillWait“ и примерите от фиг. 2. Очевидно построеното от алгоритъма дърво е много по - различно от примерното дадено на фиг. 1. Това е нормално, защото алгоритъмът работи върху ладеното му множество от примери като се стреми резултатът му да е абсолютно консистентен с всички тях и да възможно най – малък.



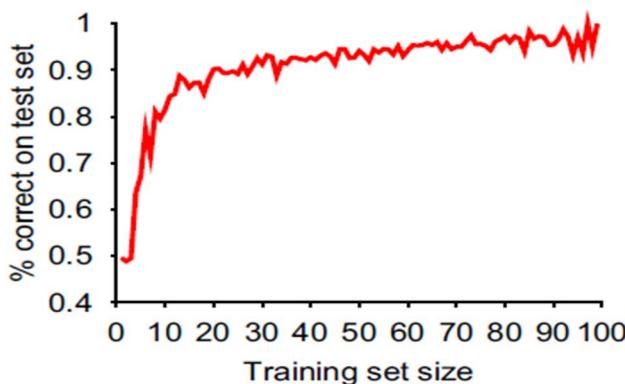
(фиг. 5)

Важно е да се отбележат и следните наблюдения:

- Алгоритъмът не е включил тестове за Raining? и Reservation?, защото е успял да класифицира всички примери без тях.
- Използвайки алгоритъмът сме открили, че ще чакаме в Тайландски ресторант в Петък и Събота.
- Също така е направил някои грешки спрямо дървото на фиг. 1 – алгоритъмът не е срешинал пример, в който ресторантът да е пълен и да трябва да се чакат 0 – 10 минути.

### Производителност на алгоритъма за строене

Точността на алгоритъмът за учене можем да оценим с *крива на ученето* като тази на фиг. 6.



(Фиг. 6)

Имаме 100 примера, които разделяме на две множества – примери за учене и примери за тестване. С примерите за учене построяваме дърво на решения (хипотеза), а с тестовите примери проверяваме резултатите, които дървото ни дава. Както се вижда с увеличаването на броя на примерите за учене, процентът точни резултати, които ни дава функцията също се повишава.

### Избиране на „най – добрия“ атрибут

Лакомото търсене, използвано при ученето по дърво на решенията, минимизира дълбочината на резултантното дърво. Идеята е да се изберат входните атрибути, които максимално класифицират входните примери. Перфектният атрибут разделя примерите на множества, в които всички примери са само положителни/отрицателни.

Всичко, което ни остава е да формализираме какво означава „най – добрият“ атрибут и ще можем веднага да се справим с функцията *Choose-attribute*. За целта ще използваме **ентропията** – мярка на несигурността на случайна променлива. Колкото по – малка е ентропията толкова по сигурна е информацията, която имаме:

- Променлива, която може да приема само една стойност има ентропия 0, тий като тя винаги ще бъдед тази стойност и няма нищо несигурно.
- Променлива с две стойности, които се падат с равна вероятност, има 1 бит ентропия. Пример е хвърлянето на монета, при което има равна вероятност да се паднат и двете страни.
- В случай, че мягтаме монета, едната страна на която се пада в 99 % от случаите, а в останалия 1 % се пада другата, очевидно трябва да има ентропия близка до 0.

Говорейки общо ентропията на случайна променлива  $V$  със стойности  $v_1, v_2, \dots, v_k$  всяка с вероятност да се падне  $P(v_k)$  се дефинира по следния начин:

$$H(V) = -\sum P(v_k) \log_2 P(v_k)$$

Така например за една променлива с две равновероятностни стойности получаваме:

$$H(V) = -(0.5 \log_2 0.5 + 0.5 \log_2 0.5) = 1$$

За удобство ще дефинираме  $B(q)$  като ентропията на Булева случайна променлива с вероятност за истина  $q$ :

$$B(q) = -(q \log_2 q + (1 - q) \log_2 (1 - q))$$

Нека се върнем към ученето на дърво на решенията. Ако имаме  $p$  позитивни примера и  $n$  негативни примера, то вероятността да имаме лъжа е  $n/(n+p)$ . Тогава ентропията на целта е:

$$H(Goal) = B(n/(n+p))$$

Един атрибут  $A$  с  $d$  различни стойности разделя множеството от примери  $E$  на  $E_1, E_2, \dots, E_d$  подмножества. Нека всяко от тези подмножества има по  $pk$  позитивни и  $nk$  негативни примера. Така получаваме, че ентропията на този клон на дървото е  $B(nk/(nk+pk))$ . Вероятността случайно избран пример да има  $k$ -тата стойност на атрибута е  $(nk+pk)/(n+p)$ . Очаквания остатък от ентропията след тестване на

атрибута **A** е:

$$\text{Remainder}(A) = \sum \{(nk + pk) / (n + p) * B(nk / (nk + pk))\}$$

Тогава очакваната полза от теста на атрибута **A** е:

$$\text{Gain}(A) = B(n / (n + p)) - \text{Remainder}(A)$$

Всъщност точно функцията **Gain** (функция на печалбата) ни трябва, за да се имплементира *Choose-attribute* – на всяка стъпка трябва да избираме атрибута с най – голяма стойност за **Gain** (най – малка оставаща ентропия след теста).

### Пренагаждане

За някои проблеми алгоритъмът **Decision-Tree-Learning** генерира големи дървета, когато всъщност няма модел за откриване. Например, нека предвиждаме дали на хвърлен зар ще се падне 6 и входните атрибути да бъдат цвета на зара, теглото му и времето, когато хвърлянето е направено. Ако хвърляме нормално зарче естественото дърво на решението е такова със само 1 възел, който е False. Алгоритъмът обаче ще се хване и за най – дребните модели в примерите, които му се дадат и ще реши например че на 4 грамовите зарчета, хвърлени в 2 следобед се пада 6. Този проблем се нарича **пренагаждане**. Колкото повече входни атрибути имаме толкова по – вероятно е да се сблъскаме с пренагаждане.

Начинът за справяне с този проблем се нарича **отсичане** на дървото на решения. Започваме с цялото дърво на решения, получено чрез алгоритъма. След това разглеждаме възелите с тестове, които имат само листа под тях. Ако тестът ни изглежда *ненужен* – отчита само шум във входните данни – тогава го махаме от дървото и на негово място слагаме листо. Повтаряме този процес по всички възели, които имат само листа за деца.

Въпросът е как да установим, че възелът тества *ненужен* атрибут. Да предположим, че сме във възел с **p** положителни и **n** негативни примера. Ако атрибутът е *ненужен* ще очакваме, че помножествата ще имат същата пропорция на положителните към всички примери. Тогава и функцията **Gain** ще ни даде съвсем малка, близка до 0 стойност. Сега въпросът е колко малка трябва да е стойността, за да сметнем атрибута за *ненужен*.

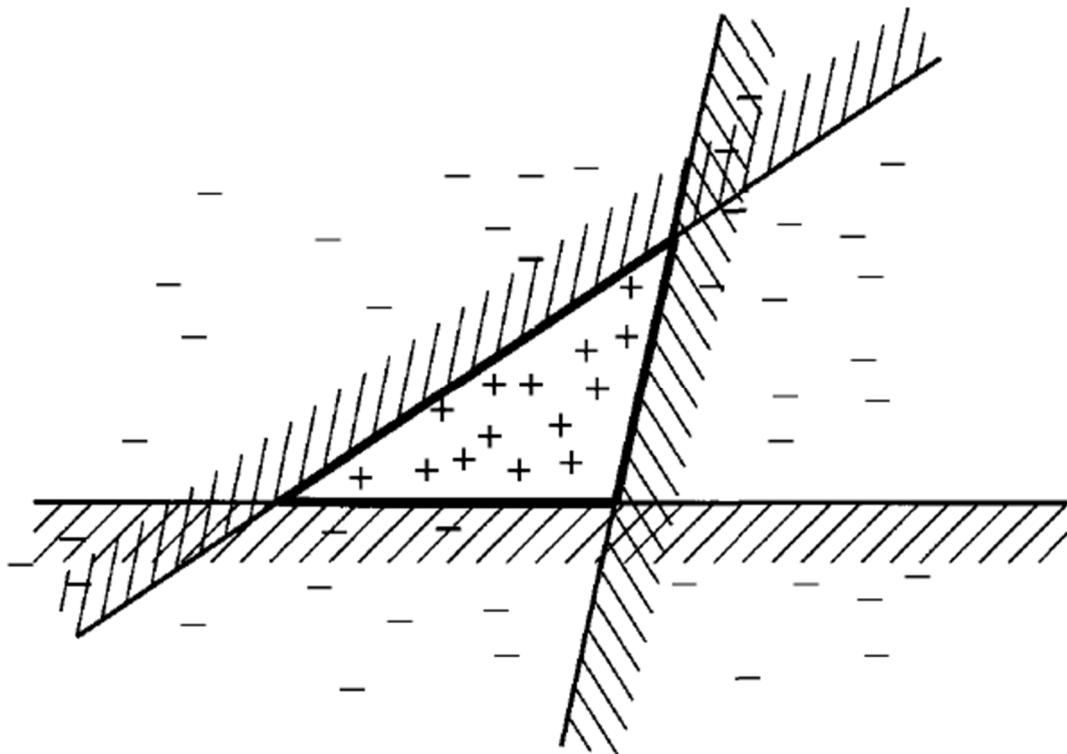
Отговорът на този въпрос може да получим чрез *статистически тест за важност*. Такъв тест започва като допускаме, че няма никакъв модел в данните (нулева хипотеза). След това анализираме същинските данни, за да изчислим до къде те се отклоняват от перфектното отсъствие на модел. Ако отклонението е статистически голямо това е добро доказателство, че имаме модел в данните.

## 13.2 Ансамблово учене

До тук разглеждахме методи за учене, които ни дават една единствена хипотеза, която използваме, за да правим предвиждания. Идеята на **ансамбловото учене** е да се избере цяла **колекция** от хипотези, след което техните предвиждания да се комбинират. Например можем да генерираме стотици различни дървета на решения от един и същи примери и мажоритарно да решим за резултата от нов пример.

Мотивацията за ансамбловото учене е много проста. Ако имаме 5 различни хипотези, от които на базата на мажоритарно гласуване решаваме какъв ще е резултатът за нов пример, то трябва поне 3 от хипотезите да събъркат, за да събъркame крайния резултат.

На фиг. 7 ясно се вижда ползата от ансамбловото учене. Взимаме 3 хипотези, всяка от които линейно отсича множеството от входове на положителни и отрицателни. Получената хипотеза ще оценява положително вход ако и 3те хипотези са го оценили положително (получения по средата триъгълник). Така получаваме много по - точна класификация на входовете.



(фиг. 7)

Един от най – широко разпространените методи се нарича **boosting**. За да разберем как работи първо трябва да изясним какво означава **претеглено множество от примери**. В такова множество всеки пример има асоциирано тегло  $w_j \geq 0$ . Колкото по високо е теглото на даден пример толкова по - важен е той по време на конструиране на хипотезата.

Boosting започва с множество от примери, всички с тегла 1 (нормално множество примери). От това множество се генерира хипотеза **h1**. Тази хипотеза ще класифицира някой от примерите правилно, а при други ще събърка. Искаме следващата хипотеза, която ще конструираме да се справи по – добре със събърканите примери. За целта увеличаваме теглата на неправилно класифицираните примери. От полученото ново претеглено множество пак конструираме хипотеза **h2**. Този процес повтаряме **M** пъти, където **M** е входен параметър. Крайната хипотеза е премерена-мажоритарна комбинация на всички **M** хипотези като теглото се изчислява на базата на това колко добре се е представила всяка хипотеза с множеството примери. Има много варианти на boosting алгоритми, които по различен начин настройват теглата и комбинират хипотезите. Един специфичен алгоритъм, наречен **AdaBoost** е показан на фиг. 8.

```

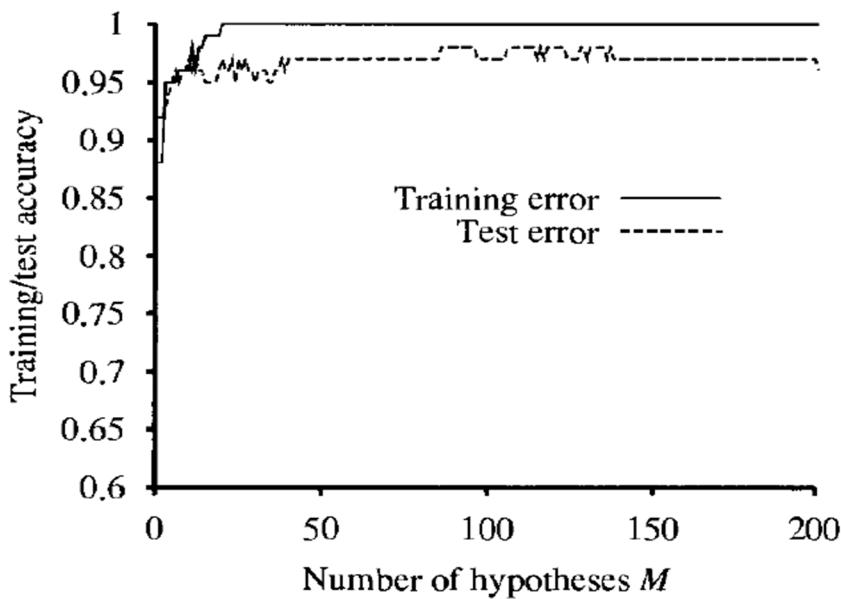
function ADABOOST(examples, L, M) returns a weighted-majority hypothesis
  inputs: examples, set of N labelled examples  $(x_1, y_1), \dots, (x_N, y_N)$ 
          L, a learning algorithm
          M, the number of hypotheses in the ensemble
  local variables: w, a vector of N example weights, initially  $1/N$ 
                     h, a vector of M hypotheses
                     z, a vector of M hypothesis weights

  for m = 1 to M do
    h[m]  $\leftarrow L(\text{examples}, \mathbf{w})$ 
    error  $\leftarrow 0$ 
    for j = 1 to N do
      if h[m](xj)  $\neq y_j$  then error  $\leftarrow error + w[j]$ 
    for j = 1 to N do
      if h[m](xj) = yj then w[j]  $\leftarrow w[j] \cdot error / (1 - error)$ 
    w  $\leftarrow \text{NORMALIZE}(\mathbf{w})$ 
    z[m]  $\leftarrow \log(1 - error) / error$ 
  return WEIGHTED-MAJORITY(h, z)

```

Фиг. 8

Макар и детайлите по смятането на теглото да не са важни, този алгоритъм има едно важно свойство: ако **L** – входният алгоритъм за учене е слаб т.е. вероятността да познае е малко по-голяма от  $1/2$ , то **AdaBoost** ще върне хипотеза, която познава всички примери от даденото му множество за достатъчно голямо **M**. На фиг. 9 можем да видим как с нарастването на **M** нараства и производителността на получената хипотеза.



Фиг. 9

На дадената графика хоризонтално се изменя стойността за **M**, а верикално е показана точността върху примерите за учене/тестване. Важно е да се отбележат 2 неща:

При  $M > 20$  резултантната хипотеза не прави нито 1 грешка върху примерите за учене.

Производителността върху примерите за тестване продължава да се увеличава дълго време след като хипотезата връща верни резултати за всички учебни примери. (максималната достигната точност върху тестовите примери е 0.97 при  $M = 135$ )

### 13.3 Речник

Prune	Отсичане
Decision tree	Дърво на решениета
Propositional logic	Съждителна логика
Divide-and-conquer strategy	Стратегията разделяй и владей
Disjunctive normal form	Дизюнктивна нормална форма
Conjunction	Конюнкция
Majority function	Мажоритарна функция
Learning curve	Крива на ученето
Mode	Мода
Significance test	Тест за важност
Null hypothesis	Нулева хипотеза
Entropy	Ентропия
Ensemble learning	Ансамблово учене
Overfitting	Пренагаждане

### 13.4 Ресурси

Artificial Intelligence: A Modern Approach (3rd Edition) Stuart Russell, Peter Norvig, 2010

Artificial Intelligence: A Modern Approach (2nd Edition) Stuart Russell, Peter Norvig, 2003

Лекции на тема “Учене на дърво на решениета.” от

<http://aima.cs.berkeley.edu/2nd-ed/>

# **14 Статистически методи за учене. Наивен Бейсов модел**

Основната цел на статистическите методи за учене е да се осигури рамка (framework) за изучаване на проблема с извеждането на заключения, т.е. за събиране на знания, правене на предвиждания, вземане на решения или конструиране на модели на базата на дадено множество от данни (примери). Тези методи са много полезни при търсенето на предсказваща функция върху дадени данни. Наивният Бейсов метод е един от най-често използваните статистически методи за обучение.

## **Съдържание**

[Речник](#)

[Статистическо учене](#)

[Наивен Бейсов модел](#)

[Използвана литература](#)

## **14.1 Речник**

Bayesian learning – Бейсово обучение

Bayesian prediction – Бейсово предвиждане

Framework – рамка

Independently and identically distributed – независимо и еднакво разпределен

Learning problem – задача с обучение

Likelihood – вероятност

Maximum a posteriori hypothesis – хипотеза на апостериорния максимум

Maximum-likelihood hypothesis – хипотеза на максималната вероятност

Minimum description length – минимална дължина на описанието

Observation – опит

Posterior probability – условна вероятност

Predictive function – предсказваща функция

Probability distribution – вероятностно разпределение

Regularity - повтаряемост

Trade-off – компромис

## 14.2 Статистическо учене

Ще обясним статистическото учене с помощта на един прост пример. Нека фирма произвежда дължи с вкус на череша и дължи с вкус на лимон. Дължите се опаковат в еднакви опаковки, така че са неразличими отвън. Продават се в чували. Съотношението на дължите в чувалите обаче е различно. Общо имаме пет вида съотношения, които ще означим с  $h_1, h_2, h_3, h_4$  и  $h_5$ , където:

$h_1$  – 100% череша, т.е. всички дължи са с вкус на череша;

$h_2$  – 75% череша + 25% лимон, т.е. 75% от дължите са с вкус на череша и 25% – с вкус на лимон;

$h_3$  – 50% череша + 50% лимон;

$h_4$  – 25% череша + 75% лимон;

$h_5$  – 100% лимон.

Нека разгледаме даден чувал с дължи. Бъркаме в чувала пъти и вадим и разопаковаме последователно п на брой дължи:  $d_1, d_2, \dots, d_n$ , където възможните стойности за  $d_1, \dots, d_n$  са череша и лимон. Основната задача на агента е да предвиди дали следващата разопакована дължа ( $d_{n+1}$ ) ще е с вкус на череша или ще е с вкус на лимон. Този проблем може да се реши с помощта на статистически метод за обучение като се използват наличните факти, т.е. разопакованите до момента дължи.

С помощта на Бейсовото обучение, на базата на дадените данни, може да се пресметне вероятността на всяка от петте хипотези ( $h_1, h_2, h_3, h_4$  и  $h_5$ ) и да се направи предположение за вкуса на следващата дължа. Това предположение се прави като се вземат предвид всички хипотези, със съответните им вероятности, а не като се вземе предвид само „най-добрата“ хипотеза. Следователно, обучението се свежда до правене на заключение на базата на вероятности.

Нека сме разопаковали дължа и тя има вкус  $d$ . Тогава вероятността на всяка хипотеза можем да изчислим по формулата на Бейс:

$$(1) P(h_i|d) = \frac{P(d|h_i)*P(h_i)}{P(d)}, i=1\div 5$$

Ключовите величини при Бейсовия подход са вероятностите  $P(h_i)$  на хипотезите и условните вероятности да се падне вкус  $d$  при всяка отделна хипотеза  $P(d|h_i)$ .

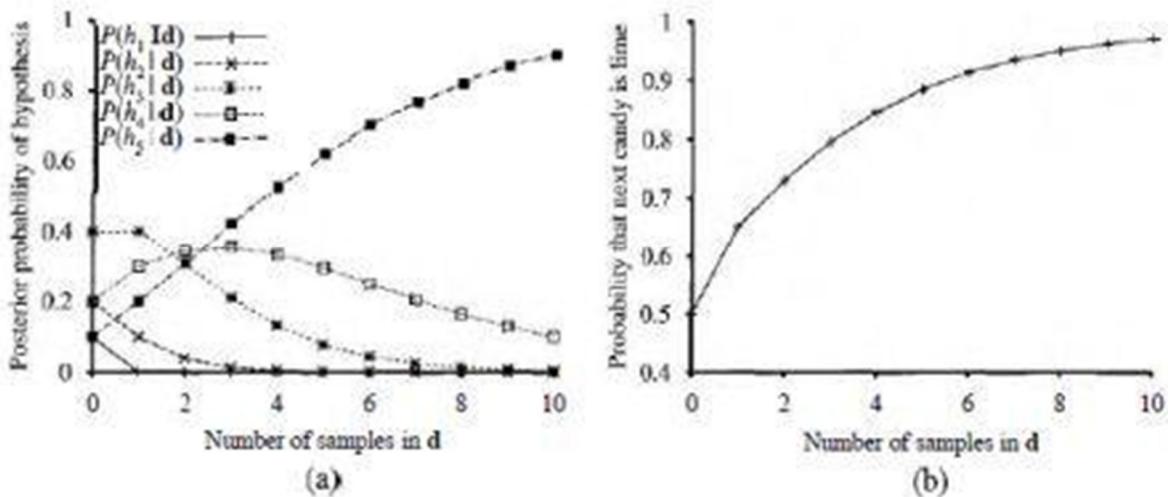
За нашия пример с дължите да предположим, че началните вероятности за хипотези  $h_1, h_2, h_3, h_4$  и  $h_5$  са съответно 0.1, 0.2, 0.4, 0.2 и 0.1. Вероятността да се падне дължа с вкус  $d$  е изчислена на базата на предположението, че опитите са независими и еднакво разпределени, така че

$$(2) P(d|h_i) = \prod_{j=1}^n P(d_j|h_i),$$

където  $h_i$  е от хипотезите  $h_1, h_2, \dots, h_5$ , а  $n$  е броят на извадените вече дължи.

Например, да предположим, че в чувала има само дължи с вкус на лимон ( $h_5$ ). Тогава първите 10 разопаковани дължи ще са с вкус само на лимон. Фигура 1 показва как вероятностите на петте хипотези се изменят с разопаковането на всяка от десетте дължи. Забележете, че в

началото вероятността за всяка хипотеза е съответната ѝ начална стойност. Следователно, в началото  $h_3$  е най-вероятната хипотеза. Това се запазва и след разопаковането на първата дъвка. След разопаковането на 2 дъвки, най-вероятна е хипотеза  $h_4$ , а след 3 или повече разопаковани дъвки,  $h_5$  е най-вероятна. На Фигура1(b) е показана вероятността следващата дъвка да е с вкус на лимон. Както и очаквахме, вероятността нараства монотонно към 1.



Фигура 1. (източник: Artificial Intelligence: A Modern Approach, Russell and Norvig, 2nd edition)

(a) Представени са условните вероятности  $P(h_i | d_1), \dots, P(h_i | d_n)$ . Броят опити  $n$  варира от 1 до 10 и при всеки опит се разопакова дъвка с вкус на лимон.

(b) Бейсовото предположение  $P(d_{n+1} = \text{лимон} | d_1, \dots, d_n)$ , където  $d_i$  означава  $i$ -тата разопакована дъвка.

Примерът показва, че вярната хипотеза в крайна сметка доминира при Бейсовото предвиждане. Това е характеристика на Бейсовото учене. Условната вероятност на всяка невярна хипотеза в крайна сметка ще се „стопи”.

Бейсовото предвиждане е оптимално както върху големи, така и върху малки множества от данни. Но тази оптималност си има цена. При истински задачи с обучение броят на хипотезите е обикновено много голям, а понякога дори безкраен.

Да предположим, че имаме неизвестна величина  $X$ . Тогава:

$$P(X|d) = \sum_{i=1}^m P(X|h_i) * P(h_i|d),$$

където  $m$  е броят на хипотезите.

В случаите, когато хипотезите са малко, тази сума може да се пресметне безпроблемно, но при много хипотези често се налага да прибегнеме до приближения и опростени методи.

Често използвано приближение е да се направят предположения на базата на една единствена хипотеза – тази, която е най-вероятна, т.е. на базата на това  $h_i$ , което има максимална условна вероятност  $P(h_i|d)$ . Това се нарича хипотеза на апостериорния максимум (maximum a posteriori hypothesis), или накратко - XAM.

Предположенията, правени на базата на хипотеза на апостериорния максимум, са приблизителни:

$$P(X|h_{\text{XAM}}) \approx P(X|d)$$

В нашия пример с дъвките  $h_{\text{ХАМ}} = h_5$  след 3 лимонови дъвки подред. Тогава ХАМ предвижда с вероятност 1.0, че четвъртата дъвка също ще е с вкус на лимон. Това е доста по-крайно предположение от Бейсовото, показано на Фигура1, което е равно на 0.8. Колкото повече опити се правят, толкова повече данни постъпват и толкова по близки стават ХАМ предвиждането и Бейсовото предвиждане. Макар нашият пример да не го показва, ХАМ обучението е доста често по-лесно от Бейсовото, защото изисква решаване на оптимизационна задача вместо намиране на голяма сума.

Хипотезата, която се използва за ХАМ осигурява максимално компресиране на данните, получени от опитите. Проблемът за намирането на хипотезата, която осигурява максимално компресиране на данните, се разглежда обаче по-директно чрез метода Минимална дължина на описанието (minimum description length), за по-кратко – МДО. МДО се базира на факта, че всяко множество от данни може да се представи чрез низ от символи от дадена крайна азбука. Основният принцип на МДО гласи, че всяка повтаряемост (regularity) в едно множество от данни може да се използва за компресиране на данните, т.е. за представяне на данните с по-малко на брой символи от необходимите за буквалното им представяне.

В случай, че безусловните вероятности  $P(h_i)$  на всички хипотези са равни, ХАМ обучението избира тази хипотеза, при която  $P(d|h_i)$  е максимално. Тази хипотеза наричаме хипотеза на максималната вероятност (maximum-likelihood hypothesis). Обучението чрез хипотезата на максималната вероятност е едно от най-разпространените статистически обучения.

### 14.3 Наивен Бейсов модел

Наивният Бейсов модел е много често използван при машинното обучение. Една от основните причини за това е, че работи вярно дори при шумове в данните. При него имаме даден клас  $C$ , а този клас, от своя страна, има екземпляри. Тези екземпляри имат определени характеристики  $x_i$ . Моделът се нарича наивен, защото предполага, че тези характеристики са независими една от друга, т.е. наличието на една характеристика по никакъв начин не влияе на наличието на останалите характеристики. Моделът се свежда до това да определим дали даден екземпляр  $X$ , който притежава определени характеристики  $x_i$ , принадлежи на класа  $C$ .

Пример: Потребителят  $X$  ще си купи ли кола като са известни неговите възраст и доход и се знае дали има шофьорска книжка?

Възрастта, доходът и наличието на шофьорска книжка са характеристиките на потребителя. Тъй като моделът е наивен, той предполага, че те са независими една от друга. Нека  $h$  е хипотеза, според която потребителят  $X$  принадлежи на  $C$ , където  $C$  е класът на хората с коли. Можем да гледаме на  $X$  като на тримерния вектор  $(x_1, x_2, x_3)$ , където  $x_1, x_2$  и  $x_3$  са съответно характеристиките на  $X$ . Нека  $P(h)$  е вероятността  $X$  да купи кола независимо от своите възраст ( $x_1$ ), доход ( $x_2$ ) и наличие на шофьорска книжка ( $x_3$ ). Нека  $P(X)$  е вероятността да се наблюдава екземплярът  $X$ . Нека  $P(X|h)$  е вероятността да се наблюдава екземплярът  $X$  при условие, че хипотезата е вярна, а  $P(h|X)$  – вероятността хипотезата да е вярна за екземпляра  $X$ . Тогава, за да решим задачата, използваме теоремата на Бейс (оттук и името на модела):

$$P(h|X) = \frac{P(X|h) * P(h)}{P(X)}$$

Нека  $D$  е обучаващо множество от екземпляри и съответните им класове. Нека всеки екземпляр  $X$  се представя като вектор от  $n$  на брой характеристики –  $X = (x_1, x_2, \dots, x_n)$ . Допускаме, че имаме  $m$  на брой класа –  $C_1, C_2, \dots, C_m$ . За да определим към кой клас принадлежи  $X$ , трябва да определим каква е вероятността  $P(C_i|X)$ . За целта използваме формулата на Бейс и получаваме:

$$P(C_i|X) = \frac{P(X|C_i) * P(C_i)}{P(X)}$$

Тъй като  $P(X)$  е константа, можем да опростим горната формула. Получаваме:

$$P(C_i|X) = \alpha * P(X|C_i) * P(C_i),$$

където  $\alpha$  е нормализираща константа.

Тъй като моделът предполага, че характеристиките са независими помежду си, можем да запишем това уравнение така:

$$P(C_i|x_1, x_2, \dots, x_n) = \alpha * P(C_i) * \prod_{j=1}^n P(x_j|C_i)$$

Веднъж обучен по този начин, моделът може да се използва за класифициране на нови примери, за които класът е неизвестен.

#### **14.4 Използвана литература**

Artificial Intelligence: A Modern Approach, Russell and Norvig, 2nd edition

Introduction to Statistical Learning Theory - Olivier Bousquet, Stephane Boucheron, and Gabor Lugosi

Wikipedia – Statistical Learning Theory

Wikipedia – Naïve Bayes Classifier

Wikipedia – Bayes' theorem

Wikipedia – Minimum description length

<http://www.scribd.com/doc/54541562/45/%E2%80%9C%D0%9D%D0%B0%D0%B8%D0%B2%D0%BD%D0%B0%E2%80%9D-%D0%91%D0%B5%D0%B9%D1%81%D0%BE%D0%B2%D0%BA%D0%BB%D0%B0%D1%81%D0%B8%D1%84%D0%B8%D0%BA%D0%B0%D1%86%D0%B8%D1%8F>

# **15 Учене без учител. Клъстерира**

Ученето без учител представлява намиране на скрити структури в некласифицирани данни. За целта има различни алгоритми, част от които ще разгледаме в темата.

## **Съдържание**

[Съдържание](#)

[Учене без учител](#)

[Клъстерира](#)

[Какво представлява клъстерира?](#)

[Техники за клъстерира](#)

[K-means](#)

[Дефиниция](#)

[Алгоритъм](#)

[Пример](#)

[Смесено Гаусово моделиране](#)

[Йерархично клъстериране](#)

[Йерархични K-means](#)

[Агломеративна техника](#)

[Речник](#)

[Източници](#)

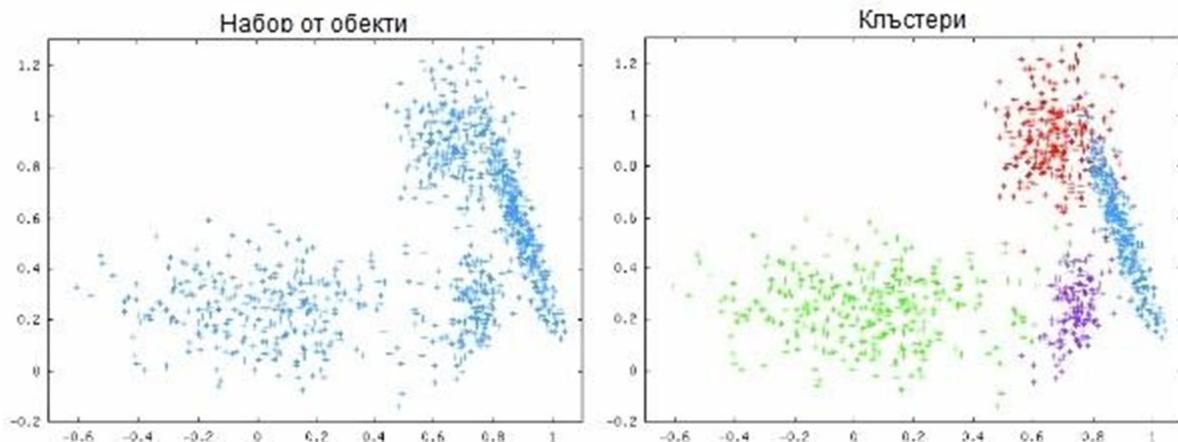
## **15.1 Учене без учител**

Ученето без учител е начин за учене, при който агентът учи модели, използвайки данни от входа, без да има предоставена изрична обратна връзка. Тоест агентът се опитва да намери скрита структура в некласифицирани данни. Тъй като данните не са класифицирани, няма сигнал за оценка, който да определи потенциални решения. Подход при ученето без учител е тъй наречената „клъстерира“ (групиране), която засича потенциално полезните клъстери от входните данни. Например, агент „такси“ може постепенно да изгради обща представа за „дни с добър трафик“ и „дни с лош трафик“, без учител да е дефинирал примери за всяко едно от тях.

### **Клъстерира**

#### **Какво представлява клъстерира?**

Клъстерира е дейността по групиране на набор от обекти, така че обектите от една и съща група (наричана клъстер), да са подобни помежду си и да се различават от обектите в останалите клъстери.



Първоначално терминологията на клъстера изглежда ясна и точна: група от сходни обекти. Но клъстерите открити от различни алгоритми могат да се различават значително по своите свойства. Така че, трябва да се има в предвид и избора на подходящ алгоритъм при разрешаването на конкретен проблем.

Техниките за клъстеризация се разделят на:

- Комбинаторни техники: работят директно върху данните
- Смесено моделиране: допуска, че данните са независими и идентично разпределени, и моделира вероятностната функция на плътността
- Търсене на режим: или “bump hunting”

Моделите за клъстеризация се разделят и на „hard“ и „soft“. При „hard“ клъстеризацията всеки обект попада в точно един клъстер. При „soft“ клъстеризацията всеки обект може да попада в повече от един клъстер като за всеки клъстер, в който попада, обектът притежава съответна степен на принадлежност.

## Техники за клъстеризация

Ще се фокусираме върху следните техники(алгоритми) за клъстеризация:

- K-means
- Смесено Гаусово моделиране (също наричано soft K-means)
- Йерархично клъстериране (Agglomerative/divisive) clustering

В практиката тези техники обикновено се използват като част от много по-голяма система, която често включва и учене с учител.

## 15.2 K-means

### Дефиниция

Сравнително прост алгоритъм. Стреми се да раздели обектите на  $k$  клъстера по следния начин:

-избират се центровете на вски клъстер

-повтаря следните стъпки докато не приключи:

\*Добавя всеки обект към кълстера с най-близък център

\*замества всеки център на кълстер със средното на всички обекти асоциирани с него

## Алгоритъм

Нека да допуснем, че данните са в двумерното пространство и са генериирани от  $k$  кълстера. Ще направим модел на проблема с  $k$  първоначални вектора, които ще наричаме *федни*. Тях ги избираме напълно произволно.

(Разпределение) Добавяме точка(обект)  $x$  към кълстер базирано на разстоянието( $x$  е добавено към най-близкото средно):

$$y = \arg \min_{k=1 \dots k} Distance(x, m_k)$$

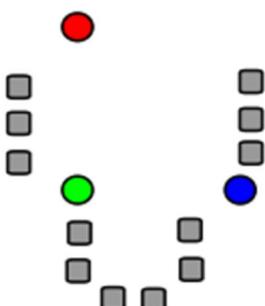
Където  $m_k$  е някой от първоначалните  $k$  вектори(средни/means).

(Обновяване) После обновяваме първоначалните състояния, като избираме средните състояния(медицентрове), базирано на точките, които сме асоциирали с всеки кълстер:

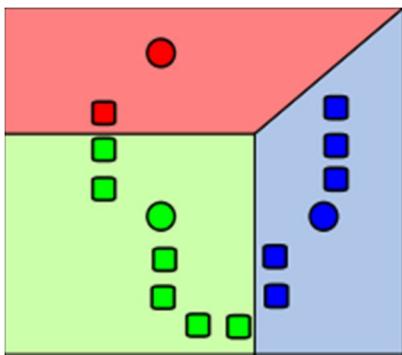
$$m'_k = \frac{1}{N_k} \sum_{x_i \text{ assigned to } k} x_i \quad k = 1 \dots k$$

$m'_k$  е новото средно(mean),  $N_k$  е броят точки, асоциирани към кълстер  $k$ , а  $x_i$  е точка, принадлежаща на  $k$ .

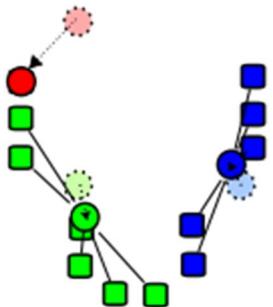
## Пример:



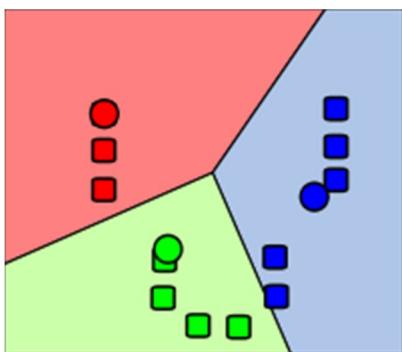
1. Произволно избиране на  $k$  кълстерни центрове(means)



2. Създаване на къмпънти, чрез асоцииране на обектите до най-близкия център.



3. Центърът на тежестта на всеки къмпънти става нов център (mean).



4. Стъпки 2 и 3 се повтарят, докато не се постигне сходство / равновесие.

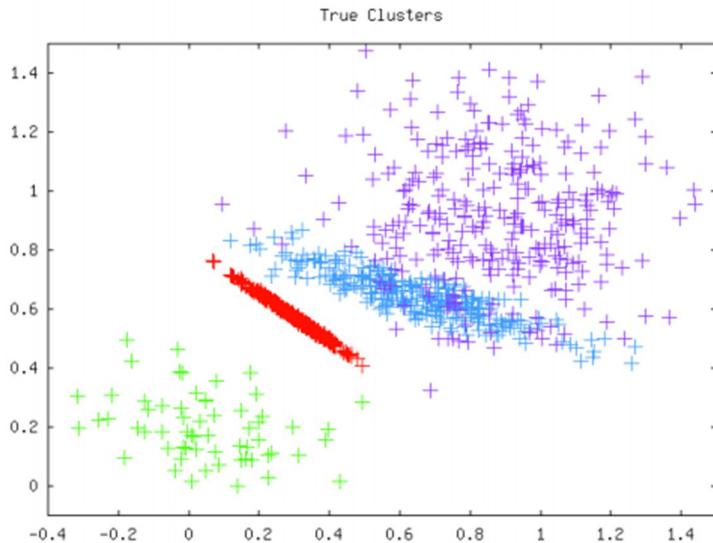
### **Как се справя с данните?**

Алгоритмът за K-means представлява техника за локално търсене за оптимизиране разпръскването на данните. Формално, K-means се опитва да оптимизира рамките на разпръскването:

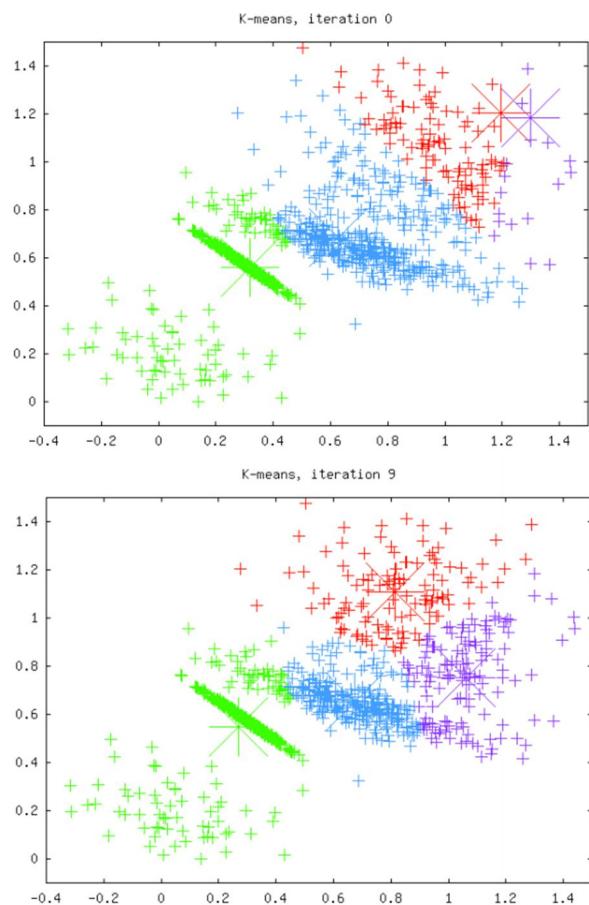
$$C = \sum_k N_k \sum_{y_i=k} ||x_i - m_k||^2$$

Тъй като това е евристичен алгоригъм, няма гаранция че ще се получи оптимално групиране и резултатът може да зависи от избора на първоначалните състояния.

Понякога се получава и зацикляне в локалния оптимум:



(истинските кълстери)



(както се вижда след 9 итерации имаме зациклияне)

Това може да се оправи чрез избиране на различни начални състояния и запомняне на най-доброто решение до момента.

Алгоритъмът обикновено е много бърз, но има случаи, при които, дори в двумерно пространство, може да отнеме експоненциално време  $2\Omega(n)$ . Но това рядко се среща в практиката.

## 15.3 Смесено Гаусово моделиране

В специален случай Смесеното Гаусово моделиране използва expectation–maximization(EM) алгоритъм, който в по-общ вариант може да се използва и при K-means. При това положение стъпката по „разпределяне“ се явява „expectation step“, а стъпката по „обновяване“ - „maximization“. И двата алгоритъма използват кълстери центрове, като по-доброто при EM алгоритъма е, че можем да имаме кълстери с различен размер и форма.

## 15.4 Йерархично кълстериране

Йерархичното кълстериране е метод, който създава йерархия от кълстери. За разлика от K-means, не е необходимо да зададем броя кълстери(K). Необходимо е само да посочим по какъв начин ще се сравняват обектите(например разстояние). На изхода получаваме дърво.

Има два вида йерархично кълстериране:

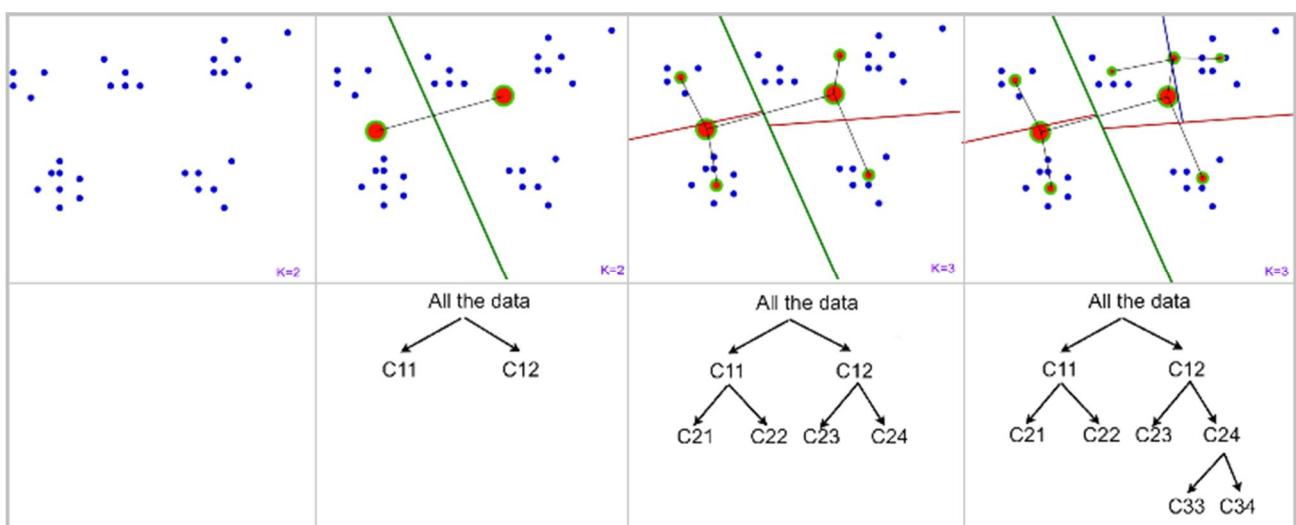
- Агломеративен: подхожда „от долу нагоре“, всеки обект стартира като собствен кълстер, а с изкачването нагоре, кълстериите се групират един с друг.
- Разделящ: подхожда „отгоре надолу“, всички обекти стартират като един кълстер, а със слизането надолу, рекурсивно се разделят.

### 15.4.1 Йерархичен K-means

Йерархичният K-means е „разделящ“ метод:

- стартира се с всички данни на един кълстер
- разделят се с помощта на „плоско“ K-средно
- рекурсивно всеки кълстер се разделя
- K обикновено е малко число
- трябва да се реши кога да спре

Пример:



(както се вижда накрая получаваме дърво от кълстери)

### 15.4.2 Agglomerative техника

Работи в обратна посока – от долу нагоре.

- При дадени N точки и начин за сравняване, се започва като всяка точка отива в отделен клас
- Открива най-близките две групи и ги слива, като това се повтаря N-1 пъти

*Начини за измерване на различията между групи:*

Означаваме разликата между две двойки данни от два различни кълстера с  $d_{ij}$ . А дистанцията между две групи G<sub>1</sub> и G<sub>2</sub> с: d<sub>SL</sub>, d<sub>CL</sub>, d<sub>GL</sub>. Тогава минималната дистанция(Single linkage) между елементите на двета кълстера е:

$$d_{SL}(G_1, G_2) = \min_{i \in G_1, j \in G_2} d_{ij}$$

Максималната дистанция(Complete linkage) между елементите е:

$$d_{CL}(G_1, G_2) = \max_{i \in G_1, j \in G_2} d_{ij}$$

Средното разстояние(Group Average) пресмятаме така:

$$d_{GL}(G_1, G_2) = \frac{1}{N_{G_1} N_{G_2}} \sum_{i \in G_1} \sum_{j \in G_2} d_{ij}$$

Ако данните са добре кълстериизирани, няма значение кое от горните ще използваме. Но ако данните не са добре кълстериизирани, ще се получат различни кълстери. При Single link ще се получат по-некомпактни кълстери(с повече разклонения), при Complete link ще са по-компактни, а при Group Average – резултатите са средни.

### 15.5 Речник

Mode seeking	търсене на режим
K-means	К-средни
Gaussian Mixture modeling	Смесено Гаусово моделиране
Hierarchical clustering	йерархично кълстериране
Combinatorial techniques	комбинаторни техники
Mixture modeling	смесено моделиране
Pdf	функция за нормална вероятностна плътност
Local search technique	техника за локално търсене

Divisive	разделяща
Agglomerative	агломеративна
	„Групи“ и „клъстери“ се използват като синоним

## 15.6 Известници

Artificial Intelligence, A Modern Approach - 2nd Edition

Лекции по Изкуствен интелект, ФМИ, зимен семестър 2012/2013

<http://en.wikipedia.org/wiki/K-means>

[http://en.wikipedia.org/wiki/Data\\_clustering](http://en.wikipedia.org/wiki/Data_clustering)

[http://en.wikipedia.org/wiki/Unsupervised\\_learning](http://en.wikipedia.org/wiki/Unsupervised_learning)

[http://en.wikipedia.org/wiki/Hierarchical\\_clustering](http://en.wikipedia.org/wiki/Hierarchical_clustering)

## **16 Учене основано на примери. Разсъждения**

### **основани на подобни случаи.**

Методът „учене основано на примери“ използвани алгоритми като nearest neighbor and locally weighted regression е един от най-директните подходи за апроксимирани на функции насочени към реални или дискретни стойности. Когато нов пример е наличен, множество от подобни примери са изкарват от паметта и се използват за да класифицират ново получение. Една основна разлика на този подход спрямо другите е, че той може да построи различни апроксимации за избраната ни функция, за всеки един различен пример, който трябва да се класифицира. Това има голямо преимущество, когато функцията ни е много сложна, но все пак може да бъде обяснена от колекция от по-лесни апроксимации.

#### **Съдържание**

Съдържание

Какво е „учене основано на примери“?

k-nearest neighbor algorithm

    Основна Информация

    Алгоритъм

    Параметри

    Свойства

    Пример

Разсъждения основани на подобни случаи

    Основна информация

    Алгоритъм

        ID3 алгоритъм

        Основна информация

        Алгоритъм

        Пример

#### **16.1 Основни Понятия и съкращения**

1. Учене основано на примери – Instance Based Learning
2. K-nearest neighbor algorithm - K-NN
3. Machine learning – машинно обучение
4. Разстояние на Хеминг - Разстоянието на Хеминг между два равноразредни вектора е броят на различаващите се компоненти. Тъй като показва броя на компонентите,

които трябва да се променят (коригират), за да се получи от единния вектор другия, има важно приложение в теорията на шумозащитните кодове.

5. Мажоритарен вот – вот в който победителя е този с най-много гласове
6. Шансът за грешка на Бейс (англ. Bayes error rate) - е най-ниският шанс, който е възможен при класификация на някакъв клас.
7. Дърво на решението – decision tree.
8. Ентропия на Шанън (ентропи, англ Entropy) – Ентропията на Шанън, която понякога се нарича и „липса на информация“, в статистическата механика е еквивалентна на ентропията. Повече може да прочетете в Wikipedia:  
[http://en.wikipedia.org/wiki/Entropy\\_%28information\\_theory%29](http://en.wikipedia.org/wiki/Entropy_%28information_theory%29)
9. Разсъждения основани на подобни случаи - Case-based reasoning (CBR)

## **16.2 Какво е „учене основано на примери“?**

Учене основано на примери в машинното обучение е семейство от учащи се алгоритми, които използват досегашните знания от паметта за да ги сравнят с новите знания. Това е един вид мързеливо учене. Това се нарича „учене основано на знания“, тъй като взима хипотезите директно от вече вкараните в паметта примери. Това означава, че сложността на хипотезите може да расте с нарастването на данните. В най-лошия случай, хипотезата е лист от  $n$  примера, и изчислението за класифицирането на нов запис е със сложност  $O(n)$ . Едно предимство на този подход е че възможността да се адаптира към не използвани преди данни. Другите подходи се нуждаят да класифицират отново всичките данни когато се добави нов пример. Един подходящ алгоритъм за този метод е k-nearest neighbor.

## **16.3 K-nearest neighbor algorithm**

### **Основна информация**

При разпознаването на шаблони, k-NN е метод за класифициране на обекти базирано на най-близките примери в пространството. K-NN е алгоритъм използван в групата от алгоритми „учене основано на пример“ или мързеливо учените алгоритми, където функцията се аппроксимира само локално и всичките изчисления се изчакват до класификацията. Алгоритъма е почти най-лесният алгоритъм от всички в машинното обучение. Обекта се класифицира спрямо вотът на повечето от мажоритарния вот на неговите съседи и се присъединява към класа, който е най-често срещан сред неговите k съседа (k е положително число). Ако  $k = 1$ , то тогава обектът просто се присъединява към класа на най-близкия му съсед.

Същият метод може да се използва и в регресионен анализ, като просто припишем свойствата на обекта да бъдат средните стойности на неговите най-близки k съседи, ако се използват стойности от истинския свят. Ако стойностите са дискретни, то тогава просто взимаме най-често срещаната стойност сред най-близките k съседи. Той може да е полезен като се използва така че най-близките съседи се взимат с по-голяма тежест, отколкото по-

далечните. Съседите се взимат от множество от обекти, в което вече правилната класификация се знае. Това множество може да се възприема като базовото множество за алгоритъма. K-nn алгоритъма е чувствителен към структурата на локалните данни.

## **Алгоритъм**

Основните примери са вектори в много мерно пространство, което съдържа различни пространства, като всяко има собствен клас. Тренировачната фаза на алгоритъма се състои само от събирането на векторите и класовете на първоначалните примери.

Във фазата за класификация  $k$  е константа определена от потребителя и новия вектор, който все още не е класифициран, се класифицира спрямо класа, който е най-често срещан сред неговите  $k$  най-близки основни примери.

Обикновено евклидово разстояние се използва за да се определи разстоянието от някаква точка до нейните съседи. Това може да се използва единствено при непрекъснатите променливи. Да речем в случаи, в които се използват класификация по текст може да се използва никакви метрики на препокриване (например Разстояние на Хеминг). Обикновено точността на класификацията може да се подобри осезаемо ако се използват метрики за разстоянието се правят с специализирани алгоритми като Large margin nearest neighbor. Недостатъка на простото мажоритарно гласуване, че най-срещаните класове ще доминират в отгатването на клас на новия вектор тъй като те са склонни да дойдат в  $k$  най-близките съседи, когато съседите се изчисляват, тъй като те са най-много. Един начин да се избегне това е като слага никакво тегло спрямо, което се взима в отсчета и най-близките съседи да се взимат с най-голяма тежест. Друг начин да се избегне този недостатък е да се вика ново ниво на абстракция в данните.

## **Избор на параметрите**

Най добрият избор на  $k$  зависи от данните. Голямото  $k$  редуцира ефекта от шума при класифицирането, но прави границите между класовете по-малки. Добро  $k$  може да се избере с различни евристични подходи, например cross-validation. Специалния случай в който класът се избира от най-близкия му съсед (т.е когато  $k = 1$ ) се нарича the nearest neighbor algorithm.

Точността на k-NN алгоритъма може да бъде чувствително намалена от наличието на шум или от неуместни свойства, или ако разликата в стойностите на някоя стойност се променя неравномерно. Много усилия са били вложени в изследването на избирането на правилни свойства и техните стойности, за да се подобри класификацията.

При проблема на бинарна класификация (когато имаш два класа) е полезно да се избира  $k$  да е нечетно число, тъй като това предотвратява равните вотове. В този случай един от известните методи за избиране на  $k$  е bootstrap method.

## **Свойства**

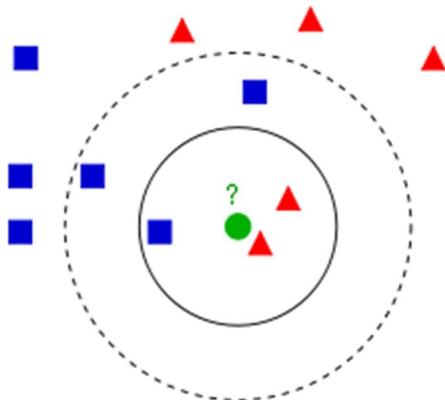
Простата версия на алгоритъма, може лесно да се имплементира, като се изчисли разстоянието от тестовия вектор до всичките вектори в паметта, но това е интензивно изчисление, особено когато данните нараснат. Много nearest neighbor search алгоритми са били предложени през годините. Главната им цел е да редуцират броя на изчисления на

разстоянията, които се извършват. Използването на правилен алгоритъм за изчисление на разстоянието прави k-NN гъвкав дори за много данни.

k-NN алгоритъма има доста силни резултати. Ако данните, които има програмата клонят към безкрайност, то точността на изчислението е много висока. Шанса за грешка е не-по-голям от двойно шансът за грешка на Бейс.

### **Пример**

Една примерна картичка на k-nn алгоритъм:



Тестовия вектор е зеленото кръгче. Той трябва да бъде класифицира или към класът на сините квадратчета или към класа на червените триъгълничета. Ако изберем  $k = 3$  (кръгчето със солидна линия), то ще получи класа на червените триъгълничета, тъй като те са 2 и има само 1 квадратче. Ако обаче  $k = 5$  (пунктиралото кръгче), то той ще получи класа на сините квадратчета, тъй като има 3 квадратчета и 2 триъгълничета.

## **16.4 Разсъждения основани на подобни примери**

### **Основна информация**

Разсъждения основани на подобни примери е процес на разрешаване на нови проблеми основани на решения на подобни проблеми от миналото. Да речем авто механик, който поправя двигател след като вече е поправил друга кола с подобни проблеми използва разсъждения основани на подобни примери. Друг пример е адвокат който оправдава клиент основан на подобен случай от миналото. Идеята на разсъждения основани на подобни примери е да се направи аналогия с подобни случаи и да се реши проблема подобно.

Спори се че CBR не е единствено мощен метод за компютърно разсъждане, но и също така механизъм, но и широко разпространение поведение в решаването на всекидневните проблеми на хората или още по радикално, че всичките разсъждения са основани на предишния опит на човек. Този поглед е свързан с prototype theory, която е по-широко изследвана в cognitive science.

### **Алгоритъм**

Процеса за изпълнение на CBR за целите на компютърното разсъждане е описан с 4 стъпки:

- Извличане:** Според дадения проблем от паметта се извлича информация за подобни случаи, които биха помогнали да се реши. Случаите съдържат в себе си проблем, неговото решение и анотация как решението е било произлязло. Да речем Иван иска да направи боровинкови палачинки. Той е новак готвач и най-подходящото решение за него е да си припомни как той някога е направил успешно нормални палачинки. Процедурата за правене на обикновени палачинки заедно с обосновките, които той е ползва за да стигне до това решение се съдържат в „случай“, който използва Иван.
- Преизползване:** Свързва се решението, което се е използвало за решаване на предишния проблем със сегашния проблем. Това може да включва преобразяване на решението за да се адаптира към новия проблем. В примера с палачинките, Иван трябва да добави боровинки, за да постигне новото решение.
- Ревизиране:** Когато нагодите старото решение да работи с новия проблем, трябва да се тества решението в истинския свят (или да се симулира) и ако е необходимо да се поправи. Да речем, че Иван е адаптиран старото решение, към новия проблем и е добавил боровинки в тестото. След като го разбърква, той разбира, че тестото е станало синьо, което е нежелан ефект. Това води до ревизирането, като се добавят боровинките, чак след като тестото бъде разбъркано и се направи на палачинки.
- Запазване:** След като решението е било успешно използвано за новия проблем, запазете решението като резултат за нов случай в паметта. Иван например запомня ново откритата процедура за правене на боровинкови палачинки и това го подготвя по-добре за бъдещи случаи, в които той ще прави палачинки.

Един алгоритъм който е свързан с CBR е ID3 алгоритъма.

## **ID3 алгоритъм**

### **Основна информация**

ID3 (iterative dichotomiser 3) алгоритъма е измислен от Рес Күнлън. Той генерира дърво на решението. Алгоритъма е предшественик на C4.5 алгоритъма. Неговата цел е спрямо някакви първоначални данни да се построи дърво на решението, с чиято помощ да класифицираме атрибути на некласифициран вектор. Необходимо е да имаме някаква базова информация за некласифицирания вектор, за да може съдейки по нея и досегашната информация от паметта да се даде някакъв извод, какво би трявало да се даде като стойност на неизвестните атрибути.

### **Алгоритъм**

Алгоритъма на кратко може да се опише така:

- Вземаме всички неизползвани атрибути и им смятаме тяхното ентропии спрямо основните примери, които вече знаем.
- Избираме атрибута с най-малко ентропии.
- Правим Листо, с дадения атрибут. Ако той разпределя множеството от примери, които имаме на две половини, като във всяка една от тях, търсения атрибут има една и съща стойност, то тогава приемаме че дървото е намерило решение. Ако обаче в някоя от двете половини имаме обекти, чиято стойност на търсения атрибут се

разминава, правим ново дърво на решението с корен даденото листо и почваме от точка 1, като смятаме ентропията само за обектите в това листо (т.е. махаме вече тези, които са били в другата половина). От примерите става по-ясно.

За да пресметнем ентропията се използва функциите:

### **Entropy**

$$E(S) = - \sum_{j=1}^n f_S(j) \log_2 f_S(j)$$

И

### **Gain**

$$G(S, A) = E(S) - \sum_{i=1}^m f_S(A_i)E(S_{A_i})$$

С дадения пример ще бъде малко по-ясна самата реализация на алгоритъма и после ще обобщя и реално какво се иска да се каже с този алгоритъм.

### **Пример**

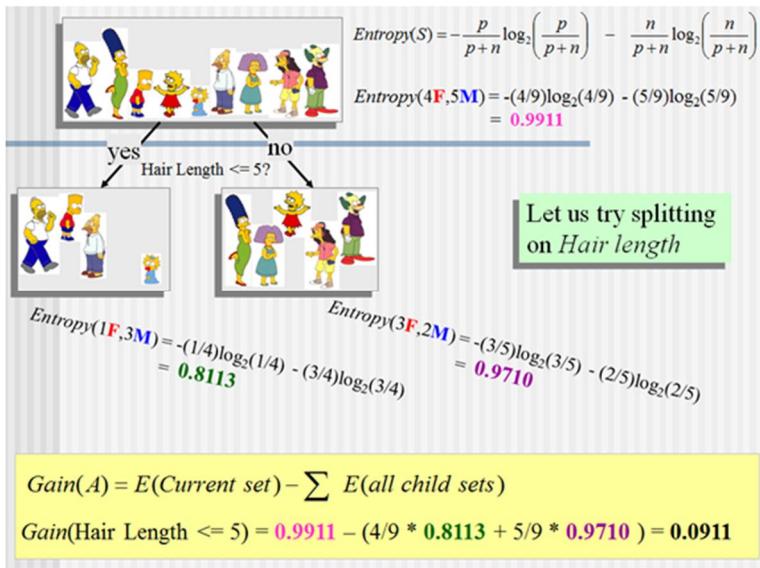
Нека имаме следната база от първоначални примери:

Person	Hair Length	Weight	Age	Class
Homer	0"	250	36	M
Marge	10"	150	34	F
Bart	2"	90	10	M
Lisa	6"	78	8	F
Maggie	4"	20	1	F
Abe	1"	170	70	M
Selma	8"	160	41	F
Otto	10"	180	38	M
Krusty	6"	200	45	M
Comic	8"	290	38	?

Използваме хората от сериала „симвъсънс“ като база и искаме да определим какъв пол е продавача на комикси. В дадения случай неизвестният клас (атрибут) е полът и той приема две стойности: M или F.

Ентропията по липсващия атрибут е 0.9911. Gain е напрактика стойността по-която определяме кой атрибут от тези, които знаем, напрактика ще изберем за корен на дървото. Той се смята като от ентропията на липсващия атрибут извадим разпределението на ентропията на атрибута, за който търсим Gain. В дадения случай да речем избираме Hair Length атрибута. Избираме (това е никакво константно число, спрямо, което мислим, че даденото разпределение трябва да е вярно) че ако дължината на косата е по-малка от пет това означава,

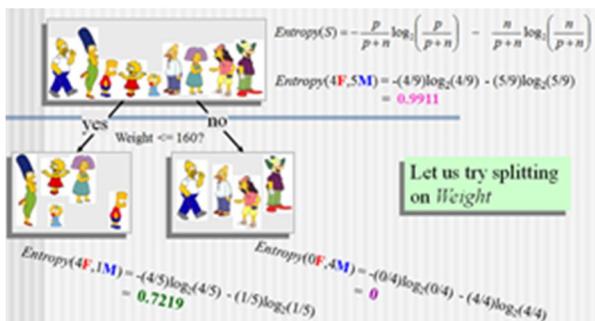
че е мъж, ако не е значи е жена. Смятаме ентропията на двете половини пак по лисващия атрибут, изкарваме го от главното ентропии и това се равнява на Gain-a:



$$Gain(A) = E(\text{Current set}) - \sum E(\text{all child sets})$$

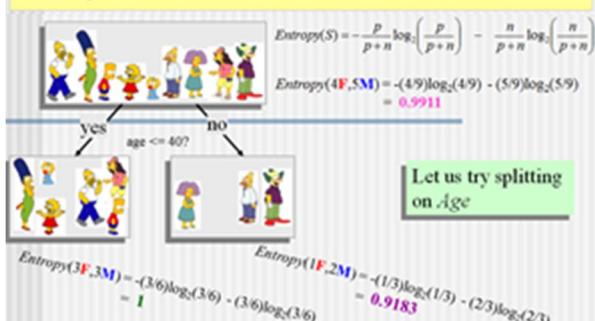
$$Gain(\text{Hair Length} \leq 5) = 0.9911 - (4/9 * 0.8113 + 5/9 * 0.9710) = 0.0911$$

Ето и сметките другите два атрибута



$$Gain(A) = E(\text{Current set}) - \sum E(\text{all child sets})$$

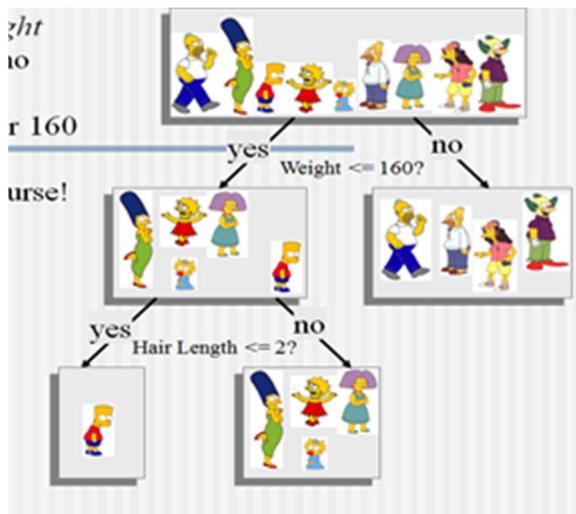
$$Gain(\text{Weight} \leq 160) = 0.9911 - (5/9 * 0.7219 + 4/9 * 0) = 0.5900$$



$$Gain(A) = E(\text{Current set}) - \sum E(\text{all child sets})$$

$$Gain(\text{Age} \leq 40) = 0.9911 - (6/9 * 1 + 3/9 * 0.9183) = 0.0183$$

Също така е важно да се внимава с логаритъма. Приемаме че при равно разпределение на класа ентропията е 1-па (както се вижда от сметката), а ако съществуват само от един от двата класа представители, то тогава ентропията е 0. Забелязваме че теглото е най-подходящия кандидат. За корен на дървото. Построяваме сега следното примерно дърво на решението, което илюстрира, как се стига до разделяне на класовете :



След като веднъж определим в кой клон ще тръгнем възприемаме, че той ни е новото множество от знания и си построяваме съответния Gain. Напрактика както казах вече, няма значение дали дълчината на косата е под 2 или под 100, въпроса е да подберем, такива цифри, така че да решим възможно най-правилно, кой в коя група пада. Напрактика може да използваме не числови стойности, а някакви номенклатурни стойности като „Дебел“, „Слаб“, „Дълга коса“, „Къса коса“. **Обобщение на алгоритъма**

Нека вземем да речем тази номенклатура:

```

@attribute outlook {sunny, overcast, rainy}
@attribute temperature {hot, mild, cool}
@attribute humidity {high, normal}
@attribute windy {TRUE, FALSE}
@attribute play {yes, no}
@data
sunny,hot,high,FALSE,no
sunny,hot,high,TRUE,no
overcast,hot,high,FALSE,yes
rainy,mild,high,FALSE,yes
rainy,cool,normal,FALSE,yes
rainy,cool,normal,TRUE,no
overcast,cool,normal,TRUE,yes
sunny,mild,high,FALSE,no
sunny,cool,normal,FALSE,yes
rainy,mild,normal,FALSE,yes
sunny,mild,normal,TRUE,yes
overcast,mild,high,TRUE,yes
overcast,hot,normal,FALSE,yes
rainy,mild,high,TRUE,no

```

Атрибутите (@attribute) са ни „параметрите“ (качества, свойства, наречете го както си искате), които има дадения обект. Да речем деня е слънчев, студен и с висока влажност. Не, той не става за игра на бейзбол. Тука обектът е деня, а негови параметри са outlook, temperature, humidity, windy, play. Всичките тези параметри обединени дават един вектор, който съдържа

информация за всяко един от тях и това напрактика ни е @data. Всеки един ред @data ни е вектор. С помощта на тези данни нашата цел е да определим състоянието на някакъв вектор, който не е напълно известен да речем ако имаме:

Sunny, cool, normal, FALSE, play = ? (Става ли за игра на бейзбол)

И тук с помощта на известните ни до сега вектори, трябва да преценим кой е най-правилния отговор. Да речем виждаме, че ако има слънце и няма вятър, то най-вероятно времето ще става за игра на бейзбол. По този начин си класифицираме неизвестния атрибут, спрямо това какво напрактика знаем, че има дадения обект(дадения ден) и как сме реагирали в досегашните ситуации при такава обстановка (дали сме играли бейзбол в предишни дни с такива параметри).

## 16.5 Ресурси

Wikipedia - [http://en.wikipedia.org/wiki/Instance-based\\_learning](http://en.wikipedia.org/wiki/Instance-based_learning), [http://en.wikipedia.org/wiki/K-nearest\\_neighbor\\_algorithm](http://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm), [http://en.wikipedia.org/wiki/ID3\\_algorithm](http://en.wikipedia.org/wiki/ID3_algorithm)

Презентация пример за ID3 алгоритъм : <http://www.cs.sjsu.edu/~lee/cs157b/ID3-AllanNeymark.ppt>

# 17 Невронни мрежи

## Съдържание

[Въведение](#)

[Биологични невронни мрежи](#)

[Основни елементи на невронните мрежи](#)

[Персепtron](#)

[Еднослоен персепtron](#)

[Многослоен персепtron](#)

[Приложения на невронните мрежи](#)

[Речник](#)

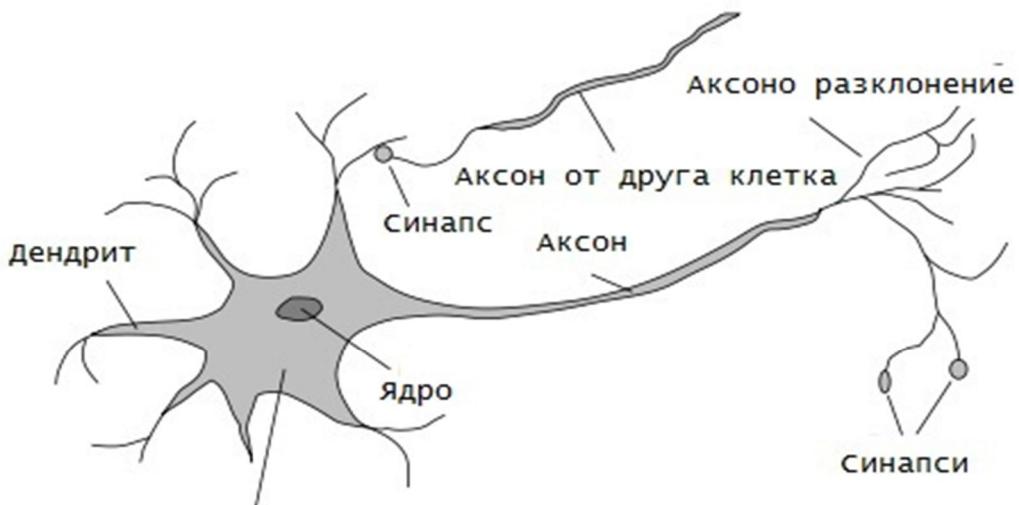
[Използвана литература](#)

## 17.1 Въведение

Видовете компютърни модели, с които работим, се основават на изкуствени невронни мрежи. Те представляват един нов подход в областта на изкуствения интелект. Този подход произлиза от разбирането, изразено за първи път преди повече от 50 години в трудовете на психолога Доналд Хеб, че човешкият мозък притежава изчислителни свойства, които се доста уникални. Тези свойства сериозно се различават от свойствата на дигиталния компютър, който има предвид метафоричното сравнение на компютър и човешки мозък. Преди около 15 години една група учени по когнитивна наука започва да изследва изчислителните следствия на тези различия. Тези изследвания довеждат до разработването на един нов подход в изкуствения интелект - невронни мрежи: подробни и правдиви модели за реални мозъчни структури, но по-често изкуствените невронни мрежи опростяват неподходящи и се опитват да обхванат по-висши и по-абстрактни свойства на невронните изчисления. Проблеми като разпознаване по образец, разпознаване на говор, разпознаване на образи и други са някои задачи, при които се счита, че невронните мрежи могат да се прилагат успешно. Без да се задълбочаваме в тази насока, ще скицираме съвсем накратко идеите, върху които се основава една невронна мрежа: това е техника за обработване на данни, вдъхновена от начина, по който това се извършва в човешкия мозък. Изкуствената невронна мрежа е математически модел, съставен от набор отделни елементи, които симулират някои наблюдавани свойства на биологичните невронни системи (както и процесите на адаптивно биологично усвояване на нови знания и умения). Съвременният модел на невронна мрежа е композиция на добре взаимодействащи си елементи и свързващите ги канали. Основно свойство на невронните мрежи е самомодификацията дотогава, докато бъде постигнат определен желан резултат.

## 17.2 Биологични невронни мрежи

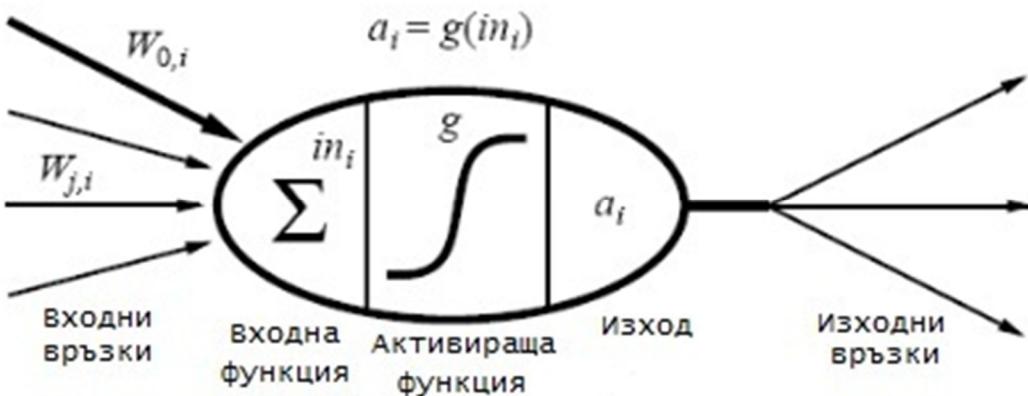
Основните градивни елементи на (изкуствените) невронни мрежи са силно опростени модели на биологичните неврони, от които е съставен човешкия мозък. За да направим аналогията с изкуствените невронни мрежи, първо ще разгледаме структурата на нервните клетки у человека. Простата невронна клетка – невронът – има тяло и разклонени структури, които осигуряват сензорният вход към неврона и извеждането на електрически сигнали от него. Дендритите са входните устройства на неврона, те управяват импулсите към невронната клетка. Аксонът управлява импулсите навън от тялото на клетката и може да се разглежда като изходното устройство на неврона. Снопове от неврони или нервни фибри формират нервната структура. В опростен сценарий нервите управяват импулсите от рецепторните органи (като очи и уши) към други органи (като мускули или жлези). Точката между два неврона, в която краят на аксона на един неврон идва в близост до тялото или до дендритите на друг неврон, се нарича синапс. В тази точка се осъществява контакта между два неврона. Импулсите (химически и електрически изменения) преминават само в една посока. Реакцията на неврона се определя от праговото ниво на чувствителността. Сигналите идват до синапса. Те са входовете и са оразмерени с никакви тегла, т.е. някои сигнали са по-силни от други. Някои сигнали активират (те са положителни), други – задържат (те са отрицателни). Ефектът на всички оразмерени входове се сумира. Ако сумата е равна или по-голяма от прага на неврона, тогава неврона реагира (извежда сигнали). Предаването на сигнали се влияе от състоянието на нервната система. Синапсите са възприемчиви към умора, кислородна недостатъчност, алкохол и др. Тези събития намаляват влиянието на импулсите. Други събития могат да усилят реакцията на неврона. Тази възможност за настройване на сигналите е механизма на научаване. Праговите функции интегрират енергията на входните сигнали в пространството и времето. На Фигура 1 е показана биологична невронна клетка.



Фигура 1

### 17.3 Основни елементи на невронните мрежи

Всяка изкуствена невронна мрежа е система от обработващи елементи, които са силно опростени модели на биологичните неврони и затова често условно се наричат (изкуствени) неврони. Обработващите елементи са свързани помежду си с връзки с определени тегла. Всеки обработващ елемент преобразува входните стойности (входните сигнали), които получава, и формира съответна изходна стойност (изходен сигнал), която разпространява към елементите, с които е свързан. Това преобразуване се извършва на две стъпки. Най-напред се формира сумарният входен сигнал за дадения елемент, като за целта отделните входни стойности (сигнали) се умножават по теглата на връзките, по които се получават, и резултатите се събират. След това обработващият елемент използва специфичната за мрежата активационна функция (функция на изхода), която трансформира получения сумарен вход в изхода (изходния сигнал) на този елемент. Основният обработващ елемент в изкуствените невронни мрежи е изкуственият неврон. Той не съответства напълно на биологичните неврони. На Фигура 2 е представен изкуствен неврон.



Фигура 2

Всеки неврон получава входен сигнал от съседни неврони или други източници и го използва, за да изчисли изходен сигнал, който предава на други възли в мрежата. Различават се три типа възли в невронната мрежа:

- входни възли, които получават данни от средата, в която работи системата(входен слой)
- изходни възли, които изпращат данни навън от системата (изходен слой)
- скрити възли, чиито входни и изходни сигнали са в мрежата (скрит слой)

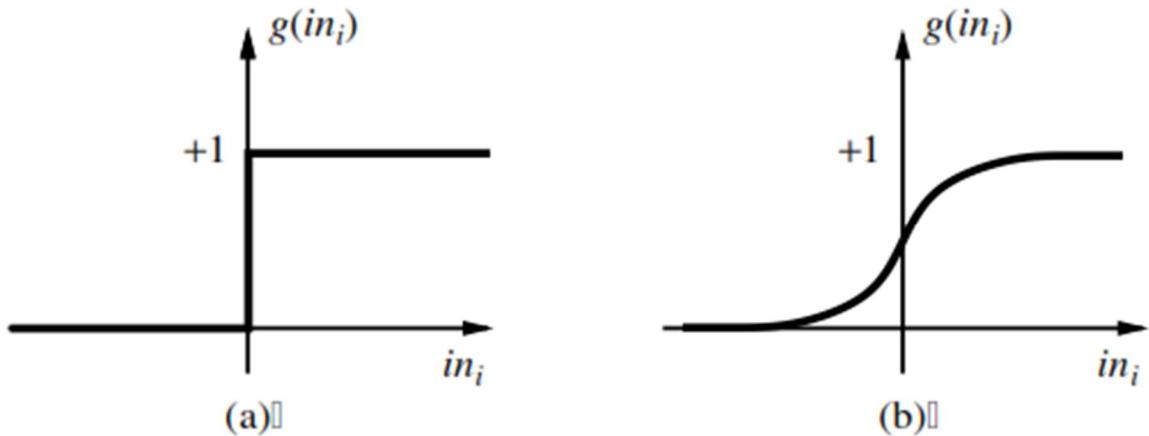
Обработката на информацията, която се извършва от неврона може да се представи със следния израз:

$$a_i \leftarrow g(in_i) = g \sum_j W_{j,i} a_j$$

където  $i$  се променя от 0 до  $n$ . С  $w$  е означен тегловният коефициент на връзката между тялото и дендритите на неврона(стойността на теглото определя силата на връзката); с  $a$  е означена

активността на невроните, свързани с неврон  $i$ ,  $a_i$  е изходът на неврона,  $g$  е предавателна функция,  $W_{ji}$  е тегловният коефициент между неврон  $i$  и неврон  $j$ . Ако стойността на теглото  $W_{ji}$  е положителна, връзката е активационна, в противен случай е поддъскаща. Обикновено предавателната функция е ненамаляваща функция на тоталния вход на даден неврон. Най-често се използва линейна, полулинейна или сигмоидна функция.

Изборът на активационна функция оказва важен ефект върху резултатите от невронните мрежи. Активационната функция представя математическа трансформация на обобщените теглови входни елементи, за да генерира изхода на неврона.



Фигура 3 (a) – функция с праг

Фигура 3 (b) – сигмоидна функция

Обикновено се използва сигмоидна активационна функция за скрития слой и линейна или сигмоидна за един от изходните слоеве. Технически погледнато активационна функция се състои от две части : комбинирана функция, която разлага всички входни елементи в единствена стойност и трансферна функция, която поставя нелинейните трансформации в обобщените елементи, за да изведе изходен елемент. Обикновено модела на данните определя формата на трансферната функция. В повечето случаи се прибягва до сигмоидна трансферна функция. Предимствата на логическата или сигмоидната функция са, че те са непрекъснати и са монотонно растящи или намаляващи, апроксимират постъпковата функция много добре и са диференцируеми върху цялата област и по този начин могат драматично да намалят изчислителната тежест на тренировката. Важно е да отбележим, че невронните мрежи, които имат скрити неврони, имат сигмоидна активационна функция и изхода на неврона е сигмоид или идентична функция и се наричат многослойен персепtron.

Сигмоидната активационна функция може да бъде представена:

$$g(x) = \frac{1}{(1 + e(-cx))}$$

Обаче като се обосновем на факта, че връщаната стойност от сигмоидната функция принадлежи на интервала  $[0,1]$ , то тази функция не може да бъде използвана в невронните мрежи за апроксимираща функция, която също може да има отрицателни стойности. За да поправим това може да използваме също двуполюсна сигмоидна функция:

$$g(x) = \frac{(1 - e(-cx))}{(1 + e(-cx))}$$

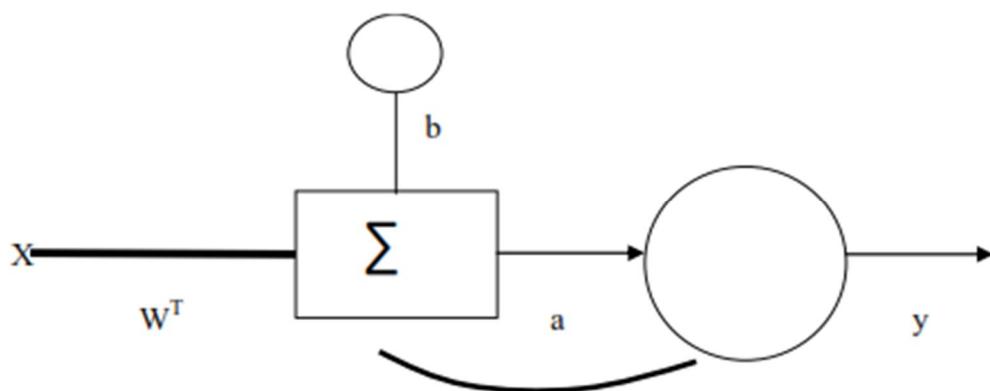
В зависимост от вида на връзките между невроните, мрежите биват:

- **прави мрежи**, в които данните се предават от входните към изходните възли без обратни връзки. Мрежата може да бъде от един или няколко слоя;
- **рекуретни мрежи**, при които има и обратни връзки. Динамичните свойства на тези мрежи са важни. При обучение невроните са в процес на релаксация, при който активностите на невроните се променят, докато мрежата достигне състояние, в което не настъпват повече значителни промени. Има набор от видими неврони, които комуникират със средата и набор от скрити неврони, които нямат подобна функция. Пример за такава мрежа е машината на Болцман.

## 17.4 Персепtron

### 17.4.1 Еднослоен персепtron

Класически пример за права мрежа е еднословийният персепtron. Това е невронна мрежа, в която всеки входен възел е свързан директно с всеки изходен възел. Еднослоен персепtron с праг представлява модела на просто обучение, който е разработен през 1962 от Розенблат. Той представлява елемент, който претегля и сумира входа, сравнява резултата с предефиниран праг. Персепtronът връща 1, ако сумата от теглата на входа е по-голяма от прага, в противен случай е 0. Беше посочва, че може би еднословийният персепtron работи като линеен дискриминант в класифицирането на елементи, които принадлежат на различни множества. За съжаление, когато става въпрос за решаването на класификационни проблеми, които не са линейно разделени, алгоритъмът на Розенблат не може да бъде завършен. Архитектурата на персептрана е показана на фигура 4.:



Фигура 4

$W^T$  = матрица от теглата

$a = b + W^T X$

$X$  = вектор от входа

$$y = f(a)$$

### **Алгоритъм за обучение на еднослоен персепtron с праг**

Нека на теглата  $w_i$  присвоим произволни начални стойности. След това с помощта на учител, който отговаря дали съответният резултат е верен или не е верен, се проверява как работи персепторът върху множество от тестови примери. За тях верният резултат е предварително известен. При това за всеки от разглежданите примери се проверява дали отговорът, даден от него за персептрона, е правилен или неправилен. Ако отговорът на персептрона за съответния пример е правилен, текущите стойности на теглата не се променят и се преминава към следващия пример. Ако отговорът е неправилен, възможни са два случая: отговорът на персептрона да бъде 1 (при правилен 0) или да бъде 0 (при правилен 1). В първия случай стойностите на  $w_i$  се коригират, като се намаляват със стойности, пропорционални на съответните  $x_i$ ; във втория случай стойностите на  $w_i$  се коригират, като се увеличават със стойности, пропорционални на съответните  $x_i$ . След съответната корекция на теглата се преминава към следващия пример.

### **Точна формулировка на алгоритъма**

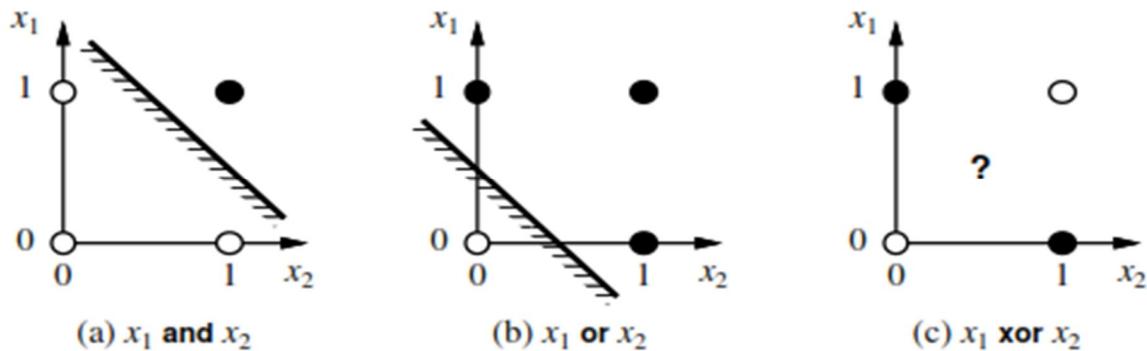
Имаме множество от обучаващи примери за задача за разпознаване (класификация) с  $n$  входни характеристики ( $x_1, \dots, x_n$ ) и два изходни класа. Търсим множество от тегла ( $w_0, w_1, \dots, w_n$ ), които са такива, че персептронът дава стойност 1 тогава и само тогава, когато входните данни съответстват на елемент от първи клас.

Стъпки за изпълнение на алгоритъма:

1. Създава се персепtron с  $n+1$  входни елементи и  $n+1$  тегла, където допълнителният входен елемент  $x_0$  винаги има стойност 1.
2. Инициализират се със случаини приближени стойности теглата ( $w_0, w_1, \dots, w_n$ ).
3. При текущите стойности на теглата  $w_i$  се класифицират примерите от обучаващото множество, след което от тях се подбират само тези, които се класифицират неправилно.
4. Ако всички обучаващи примери се класифицират правилно, като резултат от работата на алгоритъма се извеждат текущите стойности на теглата  $w_i$  и край.
5. В противен случай се пресмята вектора  $s$  като сума от неправилно класифицираните вектори  $x = (x_1, \dots, x_n)$ . При формирането на сумата  $s$  в нея с положителен знак участват всички вектори  $x$ , за които персептронът неправилно е дал резултат 0, а с отрицателен знак – тези, за които персептронът неправилно е дал резултат 1. Така получената стойност на  $s$  се умножава с предварително избран скаларен коефициент  $\eta$ . Стойността на  $\eta$  определя скоростта на доближаване до търсените стойности на  $(x_1, \dots, x_n)$  и изборът зависи от спецификата на решаваната задача.
6. Модифицират се теглата ( $w_0, w_1, \dots, w_n$ ), като към тях се прибавят съответните елементи от вектора  $s$ . Преминава се към стъпка 3.

Еднословният персепtron е добро средство за решаването на класификационни проблеми, които са линейно разделими. Може да решава проблеми основани на булевите функции(Фигура 5) - логическо И, ИЛИ, отрицание, но както споменахме не може да решава

задачи за класификация при два класа, които не са разделими линейно (например проблеми, основани на функцията XOR).



Фигура 5

Друг вид еднослоен персепtron е сигмоидният персепtron, който има сигмоидна активационна функция. Той има подобно ограничение като при еднослойния персепtron с праг, но със забележката, че той представя гъвкави линейно разделими проблеми. Ще разгледаме алгоритъма за обучение на еднослоен сигмоиден персепtron. Идеята на този алгоритъм е да коригира теглата на мрежата, така че да минимизира размера на грешката на трениращото множество. Това обучение е формулирано като оптимизационно търсене в пространството на теглата. Класическият размер на грешката е сума от квадратите на грешките. Квадратна грешка за пример с вход x и изход y е:

$$E = \frac{1}{2} Err^2 \equiv \frac{1}{2} (y - h_w(x))^2,$$

където  $h_w(x)$  е изхода на персептрана на примера. Може да използваме монотоно намаляване, за да намилим квадрата на грешката, изчислявайки първата производна на E за всяко тегло:

$$\frac{\partial E}{\partial W_j} = Err \times \frac{\partial Err}{\partial W_j} = -Err \times g'(in) \times x_j,$$

където  $g'$  е производна на активационната функция. Тогава преработваме теглата така :

$$W_j \leftarrow W_j + \alpha \times Err \times g'(in) \times x_j,$$

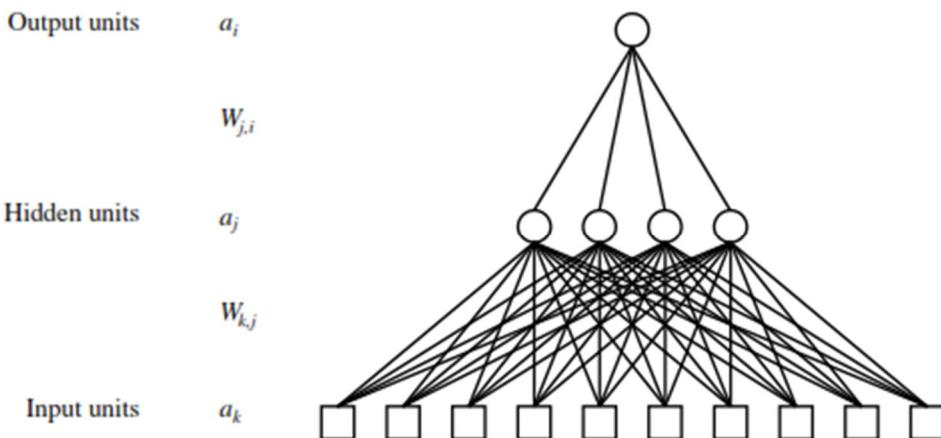
където  $\alpha$  е степента на обучение. Ако грешката  $Err = y - h_w(x)$  е позитивна, тогава изхода на мрежата е прекалено малък, така че теглата се увеличават за положителен вход и намаляват за отрицателен. Обратното се случва, когато грешката е отрицателна. Алгоритъмът подкарва примерите през мрежата един по един, като коригира частично теглата им след всеки пример, за да намали грешката. Един цикъл през примерите се нарича епоха. Епохите се повтарят докато не се изпълни някой от критерите за спиране.

## 17.4.2 Многослоен персепtron

За решаването на линейно неразделими проблеми ще използваме многослоен персепtron. Това е персепtron с поне един скрит слой, при който съществуват връзки от всеки елемент от даден слой към всеки елемент от непосредствено следващия слой. Предимството на

Добавянето на скрит слой е разширяването на множеството от хипотези, които персепtronа може да представи. За всеки скрит слой може да мислим като за еднослоен персепtron, който представя гъвка функция с праг във въдното пространство, а за изхода като линейна комбинация от няколко гъвкави функции с праг.

На фигура 6 е показан многослоен персепtron.



Фигура 6

### **Алгоритъм за обучение на персепtron с обратно разпространение**

**Идея:** Задават се случаенни начални стойности на търсените тегла, които се коригират при проверка на действието на персептрана върху всеки от обучаващите примери. Работата върху всеки от примерите се извършва на два етапа, които се наричат съответно прав и обратен пас. При правия пас се получава изходът на мрежата, съответен на текущите стойности на търсените тегла. При обратния пас получените изходни стойности се сравняват с правилния изход за съответния пример и се коригират стойностите на теглата на връзките  $w_2(i, j)$ , стойностите от елементите на скрития слой  $h_i$  и теглата на връзките  $w_1(i, j)$ .

### **Точна формулировка на алгоритъма**

**Дадено:** множество от обучаващи примери

**Търси се:** подходящи стойности  $w_1(i, j)$ ,  $w_2(i, j)$  на теглата на връзките в персепtron с три слоя, който по дадени входни стойности от обучаващото множество връща съответните изходни стойности,

**Алгоритъмът по стъпки:**

1. Нека  $A$  е броят на елементите от входния слой и  $C$  е броят на елементите от изходния слой. Избира се подходяща стойност на  $B$ , равна на броя на елементите от скрития слой. За целта има многобройни резултати, които могат да се използват. За удобство във входния и скрития слой се добавя още един по един, допълнителен елемент с постоянна стойност 1. Така индексите на елементите от входния слой приемат стойности от 0 до  $A$ , тези на елементите от скрития слой – от 0 до  $B$ , в изходния слой – стойности от 0 до  $C$ .

Въвеждаме следните означения:

$x_i$  - стойностите на елементите от входния слой

$h_i$  - стойностите на елементите от скрития слой

$o_i$  - стойностите на елементите от изходния слой

$w_1(i, j)$  – теглата на връзките между елементите от входния и скрития слой

$w_2(i, j)$  – теглата на връзките между елементите от скрития и изходния слой

2. Инициализират се теглата на връзките в мрежата. Всяко тегло получава стойност в интервала (-0.1, 0.1)

3. Инициализират се стойностите на допълнителните елементи:  $x_0 = 1$ ;  $h_0 = 1$ . Тези стойности не се променят при следващото изпълнение на алгоритъма.

4. Избира се обучаващ пример, т.е. двойка  $(x, y)$ , където  $x$  е входен вектор и  $y$  – съответен изходен вектор. Съответните стойности от  $x$  се присвояват на елементите от входния слой.

5. Разпространяват се активационните стойности от входния към скрития слой, като за целта се използва активационна функция:

$$h_i = \frac{1}{(1 + \exp(-\sum w_1(i, j)x_i))}$$

за  $j = 1, \dots, B$  и  $i = 0, \dots, A$

6. Разпространяват се активационните стойности от скрития към изходния слой

$$o_j = \frac{1}{(1 + \exp(-\sum w_2(i, j)h_i))}$$

за  $j = 1, \dots, C$  и  $i = 0, \dots, B$

7. Пресмятат се грешките на елементите от скрития слой. Нека означим тези грешки с  $\delta_2(j)$ ; всяка от тях се пресмята с помощта на изхода на мрежата  $o_j$  и правилния изход от обучаващия пример  $y_j$ :

$$\delta_2(j) = o_j(1 - o_j)(y_j - o_j)$$

за  $j = 1, \dots, C$

8. Пресмятат се грешките от елементите от скрития слой. Нека означим тези грешки с  $\delta_1(j)$

$$\delta_1(j) = h_j(1 - h_j) \sum \delta_2(i) w_2(i, j)$$

за  $j = 1, \dots, B$  и  $i = 0, \dots, C$

9. Уточняват се стойностите на теглата на връзките между скрития и изходния слой:

$$w_2(i, j) = w_2(i, j) + \Delta w_2(i, j)$$

$$\Delta w_2(i, j) = \eta \delta_2(j)$$

за  $j = 1, \dots, C$  и за  $i = 0, \dots, B$

10. Уточняват се стойностите на теглата на връзките между входния и скрития слой:

$$w_1(i,j) = w_1(i,j) + \Delta w_1(i,j)$$

$$\Delta w_1(i,j) = \eta \delta_1(j)$$

за  $j = 1, \dots, C$  и за  $i = 0, \dots, B$

11. Преминава се към стъпка 4 и се повтарят действията от стъпки 4 – 10 за всички останали обучаващи примери.

## 17.5 Приложения на невронните мрежи

Невронните мрежи имат широко приложение в различни области. Те могат да се използват за разпознаване на изображения, за разпознаване на глас. Тъй като невронните мрежи са най-подходящи за идентифициране на схеми и тенденции в данни, те са подходящи за приложения, свързани с прогнозиране като прогнозиране на продажби, контрол на индустриални процеси, проучвания на потребителето, валидизиране на данни, управление на риска и други. Също така са използвани експериментално и в медицината за моделиране на сърдечно-съдовата система. Разработеният индивидуален модел може да се сравнява с реалните физиологически измервания на пациента, за да се постави диагноза. Невронните мрежи са в състояние да комбинират сигнали от различни сензори. Могат да заучават сложни взаимоотношения между стойности от различни сензори. Те се използват за създаване на електронен нос, който регистрира миризми с приложение в телехирургия.

## 17.6 Речник

дендрити – разклонени структури, които осигуряват сензорен вход към тялото на неврона (входни устройства на неврона)

аксон – провежда електрическия сигнал от тялото на неврона до неговите синапси

синапс – точката между два неврона, в която краят на аксона на един неврон идва в близост до тялото или до дендритите на друг неврон

## 17.7 Използвана литература

М. Нишева; Д. Шипков, Изкуствен интелект, Интеграл, 1995

Лекции по Изкуствен интелект, ФМИ, зимен семестър 2012/2013

Indarnarai Ramlall, Artificial Intelligence: Neural Networks, International research journal of finance and economics, 2008

Leonardo Noriega, Multilayer perceptron tutorial, Staffordshire University, 2005

# **18 Комуникация. Използване на формални граматики за естествен език. Синтактичен анализ.**

[Увод](#)

[N-грамен модел](#)

[Определение](#)

[N-грамен модел върху символи](#)

[N-грамен модел върху думи](#)

[Изглаждане](#)

[Извличане на информация](#)

[Отговаряне на въпроси](#)

[Регулярни изрази](#)

[Автоматизирано създаване на шаблони](#)

[Релационно извлечане на данни](#)

[Синтактичен анализ](#)

[Ограничена естествен език](#)

[Вероятностна контекстно-свободна граматика](#)

[Светът на Вампуса](#)

[Една граматика за светът на Вампуса](#)

[Парсване \(извод\)](#)

[Обучаване на вероятностите за ВКСГ](#)

[Лексикализирани ВКСГ](#)

[Съгласуване по род, число и лице](#)

[Усложнения](#)

[Заключение](#)

[Речник](#)

[Използвана литература](#)

Човешкият род се отличава от другите същества по силно развитите лингвистични умения. Чрез тях хората предават информация, заповеди, предупреждения, задават въпроси и изказват мнения. Въпреки че има много качества, уникални за човеците, комуникативните умения на агент, притежаваш изкуствен интелект, са от изключителна важност, именно зато му позволяват да получава и предава информация, заповеди, да отговаря на въпроси и да се самообучава от книги и интернет.

## **18.1 Увод**

Комуникацията с компютрите се осъществява главно чрез програмни езици. Това са езици с ясно определен синтаксис, без двусмислие и ограничен брой думи. Поради това тези езици са лесни за интерпретиране от компютър.

Естествените езици - тези, на които говорят хората помежду си - са много по-сложни. Синтактичните правила не са ясно определени. Езиците еволюират - появяват се нови думи (например през 2006 е добавен глаголът "google" към английският език), стари думи изпадат от употреба, изменят се граматичните правила и т.н. Освен това едно и също изречение може да има различен смисъл в зависимост от контекста. Хората също така използват метонимии, метафори, идиоми и други изразни средства, които са трудни за разбиране от компютри.

Излиза неосъществимо да се направи пълен модел на естествен език. Затова моделите, които ще разгледаме тук, са само приближения и разчитат главно на вероятности - например пресмятат вероятността дадено изречение да е написано на даден език, вероятността някакъв текст да бъде класифициран по определен начин, вероятността дадено изречение да се интерпретира с едно или друго значение.

## **18.2 N-грамен модел**

### **18.2.1 Определение**

Един от най-простите езикови модели е вероятностното разпределение на поредица от  $N$  обекта. Обектите могат да бъдат например символи, срички, думи, словосъчетания и цели изречения. Ще разгледаме модел, използващ вероятностното разпределение на поредица от  $N$  обекта, означени с  $c_1$  до  $c_n$ . Ще записваме  $P(1:N)$  за вероятността на поредицата  $c_1 \dots c_n$ .

Един n-грамен модел може да се разглежда като Марковска верига. Вероятността за даден обект  $c_i$  зависи само от предходящите го  $n-1$  обект. Нека разгледаме  $n = 3$  за по-удобно. Тогава вероятността за обект  $c_i$  е:

$$P(c_i | 1:(i-1)) = P(c_i | (i-2):(i-1))$$

Така вероятността за цялата поредица от обекти  $c_1:c_n$  е:

$$P(1:n) = \prod_{i=1}^n P(c_i | 1:(i-1)) = \prod_{i=1}^n P(c_i | (i-2):(i-1))$$

Първоначално трябва да се пусне алгоритъма да направи статистика на някакъв текст. Текстът, от който се обучава алгоритъма се нарича *библиотека*.

### **18.2.2 N-грамен модел върху символи**

Да видим за какво може да ни послужи n-граммият модел, когато разглежданите обекти са единични символи. Такъв модел може да се използва успешно за разпознаване на какъв е език е написан даден текст. Идеята е да се умножат вероятностите на всички n-грами в текста за всеки един от разпознаваните езици. Този език, за който полученото произведение е най-голямо, е най-вероятният за дадения текст. Практиката е показвала, че може да се получи много добра успеваемост на разпознаването дори при  $n = 3$ .

Друго, за което може да се използва този модел е проверка и поправка на правопис, класифициране на текст (т.е. разграничаване между научен текст, художествен текст, поезия, формален договор и т.н.)

### **18.2.3N-грамен модел върху думи**

При n-грамен модел, изграден върху думи, броят на обектите е много по-голям от колкото при модела, изграден върху символи. Също така, моделът върху символи едва ли ще срецне буква, който е извън азбуката, докато алгоритъмът върху думи трябва да може да се справи с аума, която не разпознава (не е в речника му).

Един възможен начин за това е, докато се сканира библиотеката, всеки път, когато се срецне непозната дума, да се замества със специален символ <НЕИЗВ> и да се добавя към познатите думи. При следващите срецания на същата дума, тя се оставя както си е. При изчисляването на вероятностите <НЕИЗВ> се третира като нормална дума.

По-нататък, когато се обработва реален текст, ако се срецне непозната дума, тя се замества с <НЕИЗВ> и по този начин се пресмята вече вероятността.

Възможно е да се разшири този модел, като неизвестните думи се разделят на класове, например <ЧИСЛО>, <МЕЙЛ>, <ЕМОТИКОНКА> и т.н.

### **18.2.4Изглаждане**

При n-граммите модели се появява проблем с твърде редки поредици, или такива, които не се срецват въобще в библиотеката. Например, в английския език няма дума, която започва с "ht". Ако 3-грамен модел е пуснат върху библиотека, съставена само от думи, които са в английския речник, то триграмата " ht" ще бъде с вероятност 0. Дали наистина трябва да е така? Ако използваме така получения модел за разпознаване на език, тогава текстът "This program issues an http request" ще има вероятност 0 да бъде написан на английски.

Искаме моделите ни да бъдат гъвкави и да могат да се справят с текстове, които не са им познати. Само защото никъде не сме виждали последователността от символи " http", не трябва да твърдим, че е невъзможно тя да се срецне. Затова моделът трябва да се промени така, че поредиците, които не са били срецани в библиотеката, да имат малка ненулева вероятност. Този метод се нарича *изглажддане*. Няколко възможни реализации на изглажддането:

- Лапласово: ако при n наблюдения булевата променлива X е била false, тогава оценката за P (X = true) = 1/(n + 2).
- "Отстъпващ" модел: ако при n-грамен модел някоя поредица има твърде нисък (или нулев) брой срецания, отстъпваме към (n-1)-грамен модел за тази поредица.
- Изглажддане чрез линейна интерполяция: комбинират се 1-граммите, 2-граммите, 3-граммите, ..., n-граммите модели с различни тегла. Например за n = 3:
  - $P(i | (i-2):(i-1)) = 3P(i | (i-2):(i-1)) + 2P(i | (i-1)) + 1P(i),$
  - където  $\lambda_1+\lambda_2+\lambda_3 = 1.$

## **18.3 Извличане на информация**

### **18.3.1 Отговаряне на въпроси**

Системите за отговаряне на въпроси са подобни на интернет търсачките, но при тях заявките са под формата на въпроси, зададени на естествен език. Това означава, че системата трябва да може да анализира въпроса на потребителя, както и да може да извлече правилната информация от интернет. Да речем, че въпросът е “Кой е убил Ейбрахам Линкълн?”. Системата среща из интернет статия за Линкълн, в която пише: “*Джон Уилкс Бут промени историята на човечеството с един юризум. Той завинаги ще бъде известен като човекът, прекъснал живота на Ейбрахам Линкълн.*” За да може да използва този откъс, за да отговори на въпроса, една система трябва да е наясно, че “прекъснал живота” означава “убил”, както и че местоимението “той” от второто изречение се отнася към подлога от първото изречение. По-нататък в тази глава ще видим как може да се направи подобен анализ на информация, но сега ще разгледаме нещо друго.

Една система за отговаряне на въпроси, AskMSR, се справя с проблема по много прост начин. Тя не разбира от местоимения, нито от синоними и идиоми, но за сметка на това знае 15 вида различни въпроси, и знае как да ги препишне така, че да могат да се използват като заявка към търсачка. Например знае, че въпросът “Кой е убил Ейбрахам Линкълн?” може да се препишне като [\* уби OR убил Ейбрахам Линкълн], както и като [Ейбрахам Линкълн е бил убит от \*] и още няколко варианта. След това подава тези заявки към търсачка и извлича околнния текст от резултатите. Фразите, които се срещат най-често, се филтрират по очаквания тип (например ако оригиналният въпрос е бил “кой”, се очакват имена на хора, “кога” - година или дата, “колко” - числа или словосъчетания, които изразяват количество) и така се съставя отговор. Наистина, тази система няма да може да извлече информацията от показаната горе статия. Но тя разчита на огромността на интернет и може да си позволи да игнорира някои резултати. Така AskMSR успяла да постигне най-добрите резултати сред конкурентните системи за времето си.

### **18.3.2 Регулярни изрази**

Чрез регулярни изрази (или регекс-ове, езизи, които могат да бъдат разпознавани от крайни логически автомати) могат да се строят шаблони, с които да се извлича еднотипна информация от мрежата. Идеята е да се извлекат атрибути на някакъв обект от текст, написан на естествен език. Ето например един регулярен израз, който разпознава цена в долари или евро:

`[$€][0-9]+([.][0-9][0-9])?`

От този регулярен израз може да се направи шаблон, който да извлича цената на някакъв продукт от страница с описание му. Шаблоните обикновено се състоят от три части: префиксен регекс, целеви регекс и постфиксен регекс. В нашия случай, когато искаме да извлечем цената на продукта от страницата му в онлайн магазин, префиксния регекс би бил “цена: “, “нашата цена: “, “само за “ както и празната дума. Целевия регекс е показаният по-горе за разпознаване на цена. Постфиксният регекс е празен.

Ако шаблонът пасва само един път в страницата, тогава може лесно да се изведе желания атрибут. Ако не се среща никъде, се налага да се остави атрибута празен или да му се даде някаква стойност по подразбиране. Ако обаче има няколко съвпадения, трябва да се намери някаква стратегия за избирането между тях. Една от тях е да се наредят шаблоните по приоритети. Например в случая с извлечането на цената на продукт, най-висок приоритет биха имали шаблоните с префикс “само за” и “нашата цена:”; “цена:” е с малко по-нисък приоритет, а празната дума - с най-нисък. Друга стратегия е да се извлекат всички възможности за стойността на атрибута и да се избере някоя от тях, например да се вземе най-ниската цена, която е в рамките на 50% от най-високата. Така бихме се справили успешно с текст от сорта на “Оригинална цена: \$99.00, нашата цена: \$78.00, доставка \$3.00”.

### **18.3.3 Автоматизирано създаване на шаблони**

Възможно е да се направи система, която автоматично да създава шаблони за извлечане на информация. Започва се с няколко ръчно подадени примера, от които системата “научава” шаблони, благодарение на които после се намират още примери, от които могат да се научат още шаблони и така нататък. Един такъв експеримент е проведен през 1999 от Сергей Брин (съосновател на Google) и е започнал с 5 примера за двойката “Заглавие на книга” - “Автор”:

(“Isaac Asimov” - “The Robots of Dawn”)

(“David Brin” - “Startide Rising”)

(“James Gleick” - “Chaos - Making a New Science”)

(“Charles Dickens” - “Great Expectations”)

(“William Shakespeare” - “The Comedy of Errors”)

Тези двойки са подадени на обикновена търсачка и след това от резултатите са извадени 10 символа преди и след съвпадението, както и текста между автора и заглавието на книгата. От тези резултати са изведени най-често срещаните конфигурации и така са създадени шаблони за търсене. В случая на Брин са се получили три използвани шаблона, с които после са извлечени 4047 двойки “Заглавие” - “Автор”. От тези нови примери са създадени още шаблони и така нататък. Еventualno са извлечени над 15000 заглавия.

Един проблем с този подход е чувствителността към грешки. Един грешен пример или един грешен шаблон може да поведе след себе си много грешни резултати. Последиците могат да се ограничат като, например, нов шаблон не се приема, освен ако не успее да намери някаква част от вече намерените примери, а новите примери не се приемат освен ако не са потвърдени от поне няколко шаблона.

### **18.3.4 Релационно извлечане на данни**

Следващата стъпка в извлечането на данни са така наречените релационни системи, които могат да се справят с множество обекти и отношенията помежду им. Така например, когато видят текста “\$249.99”, те трябва да разберат не просто, че това е някаква цена, а трябва и да разберат към кой обект се отнася тя. Една такава система е FASTUS, която е предназначена главно за новинарски статии относно корпоративни сделки, сливания и придобивания. Тя може да прочете статията:

Компанията Бриджстоун Спорт изяви в петък, че е създала съвместно предприятие в Тайван с локален концерн и японски дистрибутор за производство на стикове за голф за продажба в Япония.

и да извлече релациите:

$\epsilon \in \text{СъвместниПредприятия} \wedge \text{Продукт}(\epsilon, \text{"стикове за голф"}) \wedge \text{Дата}(\epsilon, \text{"петък"}) \wedge \text{Член}(\epsilon, \text{"Компанията Бриджстоун Спорт"}) \wedge \text{Член}(\epsilon, \text{"локален концерн"}) \wedge \text{Член}(\epsilon, \text{"японски дистрибутор"})$ .

Система за релационно извлечане на данни може да бъде направена от последователност от Крайни Трансдуктори (крайни автомати, които освен входна лента имат и изходна лента). Всеки автомат получава текст като вход и превежда текста в различен, по-удобен формат, и го предава на следващия автомат. Специално FASTUS се състои от 5 нива:

- Токенизация
- Словосъчетания
- Групиране
- Разделяне на фрази
- Сливане на структури

**Първата стъпка** - токенизацията - разделя поредицата от символи на думи, числа, пунктуация и подобни елементарни елементи.

**Втората стъпка** - словосъчетанията - групира думите в словосъчетания - например “съвместно предприятие”, “Компанията Бриджстоун Спорт”, “локален концерн” и т.н. Разпознаването на словосъчетанията може да става с регулярен изрази. Например един шаблон за име на компания може да бъде:

{Компания} = “Компания | Компанията” <Думи с главна буква>+.

**Третата стъпка** - групирането - групира словосъчетанията в глаголни групи (ГГ), съществителни групи (СГ) и обозначава другите части на изречението. Тук вече за пръв път виждаме елементарен синтактичен анализ. Както стана дума по-рано, по принцип езиковите правила са много сложни, но FASTUS разбира от сравнително ограничен език, за който правилата могат да се напишат така, че да могат да бъдат разбрани от крайни автомати. Примерът по-горе би излязъл по следния начин от тази стъпка (СЮ означава съюз, а ПР - предлог):

СГ Компанията Бриджстоун Спорт  
ГГ изяви  
СГ в петък  
ГГ че е създала  
СГ съвместно предприятие

ПР в  
СГ Тайван  
ПР с  
СГ локален концерн  
СЮ и  
СГ японски дистрибутор  
ГГ за производство  
ПР на  
СГ стикове за голф  
ГГ за продажба  
ПР в  
СГ Япония

**Четвъртата стъпка** съставя фрази. При нея отново има правила, които могат да се изразят чрез регулярни изрази (и съответно обработени с крайни автомати). От тази стъпка вече се извличат данни, които да влязат в базата данни. Например правилото за съвместни предприятия гласи:

{Компания}+ {Създаване} {СъвместноПредприятие}("с | със" {Компания}+)?

**Последната стъпка** слива и навързва структурите, създадени от предишната стъпка. Например, ако следващото изречение е "Съвместното предприятие ще започне работа през януари", на тази стъпка ще се осъществи връзката между двете споменавания за съвместното предприятие и информацията ще бъде сляга.

## 18.4 Синтактичен анализ

### 18.4.1 Ограничичен естествен език

Общуването на естествен език е доста тежка задача. Затова когато се налага нещо подобно, често се ограничава използвания език до такъв, който е близък до естествения, но по-лесен за машинна обработка. Най-често езикът се ограничава по следния начин:

- ограничаване на структурата на допустимите в езика изречения, тоест налагане на известни ограничения върху синтаксиса на езика - например максимум едно подчинено изречение на сложно изречение.

- ограничаване на предметната област на езика. Това се прави, когато системата се очаква да работи в някаква тясно определена област и така се избягват двусмислици. Нещо подобно разглеждахме при релационното извличане на данни и FASTUS, който анализира език, ограничен към предметната област на корпоративните новини. Така могат да се създават правила, които могат да се справят с тази информация, която ни интересува (корпоративни слиивания, придобивания, съвместни предприятия и т.н.) - правила, които биха били твърде много и сложни, ако езикът не беше ограничен.

По-горе видяхме как FASTUS се справя извличането на информацията от такъв ограничен език. Сега ще разгледаме истински синтактичен анализ, като постепенно усложняваме езика, върху който работим.

## **18.4.2 Вероятностна контекстно-свободна граматика**

Един от популярните модели за естествен език е вероятностната контекстно-свободна граматика, ВКСГ. Граматиката е съвкупност от правила за заменяне, които определят един език като множество от допустими низове. Терминиращите символи (ауми) започват с малка буква, а нетерминиращите - с главна. Едно правило за заменяне взима един нетерминиращ символ и го замества с нула, един или повече терминиращи и нетерминиращи символи. Един нетерминиращ символ може да бъде в началото на повече от едно правило. Процесът на генериране приключва, когато останат само терминиращи символи. "Вероятностна" означава, че към всяко правило се добавя вероятност то да се случи. Например:

$\text{ГГ} \rightarrow \text{Глагол} [0.70] \mid \text{ГГ СГ} [0.30]$

означава, че една глаголна група може да се състои само от глагол (в 70% от случаите) или от друга глаголна група, последвана от съществителна група (30%).

## **18.4.3 Светът на Вампуса**

В книгите относно синтактичен анализ често се дават примери, свързани с играта "Hunt the Wumpus", написана през 1972 на БЕЙСИК. В нея играчът се намира в пещера (квадратна мрежа от полета), в която се намира и чудовището Вампус. Чудовището е кръвожадно и ще изяде агента моментално, ако двамата се озоват на едно и също квадратче от мрежата. В пещерата също така има дълбоки ями, в които агентът може да падне (и да умре), но вампусът може да се измъкне от тях. Някъде из пещерата има кучкина злато.

Ако играчът е в квадратче, съседно на вампуса, той ще усеща воня. Ако играчът е до яма, усеща полъх на вятъра. Ако е до купчината злато, забелязва блещукане.

Играчът има на разположение една стрела, която може да изстреля в произволна посока. Ако уцели вампуса, че чува неговият предсмъртен вик, иначе - нищо, но вампусът се стресва и се премества.

## **18.4.4 Една граматика за светът на Вампуса**

### **Лексикон:**

Съществително	воня[.05]   полъх[.10]   вампус[.15]   яма[.05]   човек[.05]   ..
Глагол	е[.1]   усеща[.10]   надушва[.05]   вони[.10]   смърди[.05]   ..
Прилагателно	прав[.10]   мъртъв[.05]   смрадлив[.20]   ветровит[.02]   ..
Наречие	тук[.05]   там[.05]   напред[.05]   наблизо[.02]   ..
ЛичноМестоимение	аз[.10]   ти[.03]   мен[.10]   то[.10]   ..
ОтносМестоимение	което[.40]   който[.15]   каквото[.10]   на който[.05]   ..
СобственоИме	Борис[.01]   Стоян[.01]   Бургас[.01]   ..
КолМестоимение	никой[.10]   всеки[.10]   някой[.05]   ...
Предлог	към[.20]   в[.10]   от[.05]   на[.05]   край[.10]   ..

Съюз	и[.40]   или[.40]   но[.10]   така че[.02]   тъй като[.02]  ,..
Цифра	0[.20]   1[.20]   2[.20]   3[.20]   4[.20]  ,..

Многоточието означава, че в тази група има и още възможни замествания. За да бъде в крак с еволюцията на езика, една система трябва да обновява лексикона си периодично.

### **Самата граматика:**

И - Изречение; ГГ - Глаголна група; СГ - съществителна група; ПГ - предлозна група; ОФ - фраза с относ. местоимение; ПП - прилагателни

Нетерм.	Правило	P	Пример
И	СГ ГГ	0.80	Той + усепца полъх
	И Съюз И	0.10	Той усепца полъх + и + Борис е мъртъв
	ГГ	0.10	Там смърди.
СГ	ЛичноМестоимение	0.30	Аз
	СобственоИме	0.10	Стоян
	Съществително	0.15	яма
	КолМестоимение Съществително	0.10	всяка + яма
	ПП Съществително	0.15	смрадлив мъртъв вампус
	Цифра Цифра	0.05	3 4 (ползва се за координати на квадратче от мрежата)
	СГ ПГ	0.10	смрадлив вампус + в яма
	СГ ОФ	0.05	всеки вампус + който смърди
ГГ	Глагол	0.40	вони
	ГГ СГ	0.35	усепца + полъх
	ГГ Прилагателно	0.05	смърди + лошо
	ГГ ПГ	0.10	е + в яма
	ГГ Наречие   Наречие ГГ	0.10	усепца полъх + наблизо
ПП	Прилагателно	0.80	смрадлив
	Прилагателно ПГ	0.20	смрадлив мъртъв
ПГ	Предлог СГ	1.00	в + някоя яма
ОФ	ОтносМестоимение ГГ	1.00	на който + усепца полъх

С тази граматика могат да се генерират доста изречения, например:

Стоян е в яма.

Всеки вампус който смърди е в 2 з.

Борис е в Бургас и никакой човек усеща полъх.

Ето как би се направил разбор на изречението “Всеки вампус смърди.”:

[И [СГ [КолМестоимение *всеки*] [Съществително *вампус*]] [ГГ [Глагол *смърди*]]]

Вероятността на този разбор е  $[0.80 \cdot 0.10 \cdot 0.10 \cdot 0.15] \cdot [0.4 \cdot 0.05] = 0.8 * 0.1 * 0.1 * 0.15 * 0.4 * 0.05 = 0.000024$ . Тъй като това е единствения начин да се парсне изречението, това е и вероятността на цялото изречение.

За съжаление, тази граматика свръхгенерира. Според нея са валидни изречения като “Той усеща полъх вампус” и “Той надупва аз”. Също така няма никаква представа за членуване, спрягане на глаголите и съгласуване на родовете.

#### 18.4.5 Парсване (извод, разбор)

Един ефективен начин да се направи разбор на изречението е CYK алгоритъмът (кръстен на създателите си, Cocke, Younger, Kasami). Той изисква граматиката да бъде пренаписана в CNF (нормална форма на Чомски), тоест всяко правило да бъде от вида  $X \rightarrow \text{дума}$  или  $X \rightarrow Y Z$  (доказано е, че всяка контекстно-свободна граматика може да бъде пренаписана така). CYK работи за  $O(n^3)$  и заема  $O(n^2m)$  място, където  $n$  е броят на думите в изречението, а  $m$  е броят на нетерминиращите символи в граматиката. Алгоритъмът се основава на динамичното програмиране и строи таблица, която намира за всеки подниз от изречението вероятността на най-вероятното парсване.

```
function CYK-PARSE(words, grammar) връща P, таблица с вероятности
N ← Брой думи(words)
M ← Брой нетерминиращи символи(grammar)
P ← Масив с размер [M, N, N], инициализиран 0
/* Попълване на правилото за всяка дума от лексикона */
/* X → дума [p] означава правило от grammar, което заменя X с дума, като
правилото има вероятност p */
for i = 1 to N do
    for всяко правило от вида (X → дума [p]) do
        P[X, i, 1]←p
    /* Комбиниране на правила от тип X → Y Z */
    for length = 2 to N do
        for start = 1 to (N - length + 1) do
            for len1 = 1 to (N - 1) do
                len2 ← (length - len1)
                for всяко правило от вида (X → Y Z [p]) do
                    P[X, start, length] ← MAX(P[X, start, length],
                    P[Y, start, len1] * P[Z, start + len1, len2] * p)
return P
```

Алгоритъмът СУК не разглежда всички възможни изводи на изречението, а работи само с този, който е най-вероятен. С малко работа обаче, могат да бъдат намерени всички възможни парсвания (с експоненциална сложност).

#### **18.4.6 Обучаване на вероятностите за ВКСГ**

Това, което различава ВКСГ от нормалната контекстно-свободна граматика е атрибуутът вероятност към всяко правило. Въпреки, че може ръчно да се сложат тези вероятности, много по-добре би било те да се “научат” от реални примери. Най-лесно става това ако имаме библиотека от правилно парснати изречения. Библиотеката на Пен (Penn treebank, Marcus *et al.*, 1993) е една от най-известните такива библиотеки и се състои от 3 милиона думи, които са обозначени като части на речта и изречения, които са комплектовани с дървета на извод. Така чрез просто преброяване и изглажддане могат да се намерят вероятностите за всяко правило.

Съществуват начини да се изведе ВКСГ от необработена библиотека от изречения, но е доста по-сложно. Един алгоритъм, който прави това се нарича *inside-outside algorithm* и работи със сложност  $O(n^3m^3)$ , където  $n$  е броят думи в изречение, а  $m$  е броят на граматични групи (категории). Този алгоритъм обаче, както и алтернативите му, са прекалено бавни и имат прекалено големи изисквания за памет, за да бъдат използвани за реални случаи. Освен това граматиките, които се получават, са често твърде неудобни за използване от хора.

Възможно е да се подобри представянето на такива алгоритми, ако се започне от някаква ръчно направена, по-простичка граматика - например като тази, показана за света на вампуса. От тях, могат да бъдат след това автоматизирано научени нови правила. Също така полезно е да могат да се доизпипват старите правила, или да се разделят категории - примерно да се направи разделение на местоименията в именителен падеж (аз, ти, той) и тези във винителен падеж (мен, теб, него).

#### **18.4.7 Лексикализирани ВКСГ**

Проблемът с ВКСГ е, че са контекстно-свободни - тоест разликата в  $P(\text{"ям някакъв банан"})$  и  $P(\text{"ям някакъв асфалт"})$  зависи само от  $P(\text{"банан"})$  и  $P(\text{"асфалт"})$  - тоест колко често се е срещала всяка от двете думи в библиотеката, от която се е обучавал модела. Хората обаче знаят, че асфалта не се яде и поради това е далеч по-малко вероятно да се срещне второто изречение.

Една от възможностите за справяне с този проблем е да се лексикализира граматиката - тоест да се направи така, че вероятността на дадено правило да зависи от релациите между думите на цялото изречение. Разбира се, твърде трудна задача е да се набавят и складират данни така, че да се направи статистика за зависимостите между всяка една дума в изречението. Затова е полезно да се добави атрибут “членна дума” към граматичните групи. Така например члената дума на ГГ “ям” е *ям*, на СГ “някакъв банан” - *банан*, и на СГ “някакъв асфалт” - *асфалт*.

Ще означаваме членната дума в скоби след името на граматичната група - например СГ(банан): “някакъв *банан*”.

Тъй като граматиката вече не е контекстно-свободна, използваме DCG (definite clause grammar) за описането ѝ. Ето един пример за правило за СГ на лексикализирана граматика:

$\text{СГ}(c) \rightarrow \text{ПП}(n) \text{ Съществително}(c) \{ \text{Съвместимост}(n, c) \}.$

Новостта тук е означението {ограничение} - това означава, че правилото важи единствено и само ако ограничението е истина. Предиката Съвместимост трябва да е истина тогава и само тогава, когато прилагателното *n* може да се използва за съществителното *s*. Например Съвместимост(“черно”, “куче”) = true, докато Съвместимост(“ромбоидно”, “куче”) = false. Допълнително може да се добави функция-модификатор на вероятността *p* на правилото в зависимост от това колко са съвместими думите. Например словосъчетанието “въоръжено куче” е принципно възможно, но не много вероятно.

Подобни взаимоотношения могат да се направят и за двойките глагол - наречие, глагол - допълнение и т.н. Думите могат да бъдат разделени на категории, например категория за съществителни- храни, съществителни-превозни средства и така нататък. Това би спомогнало по-лесната поддръжка на предиката за съвместимост и функцията за вероятност, като например всички съществителни- храни се третират по един и същи начин от двойката глагол-допълнение, когато глаголът е някаква форма на “ям”.

#### **18.4.8 Съгласуване по род, число и лице**

В граматическите правила на много езици някои думи се изменят в зависимост от рода, бройката на обектите (един или много), и падежа (дали обекта е извършилелят на действието или потърпевшия, както и още няколко разновидности). Да се спазват тези правила е от голямо значение за успеваемостта на системата.

Да разгледаме първо падежите при личните местоимения в българския език. Граматиката за света на вампуса може да генерира граматически грешното изречение “Той надушва аз.” Проблемът е, че личното местоимение “аз” е в номинативен падеж, а трябва да бъде във винителен (“мен”). Това изречение има дърво на извода:

[И [СГ [ЛМестоимение *той*]] [ГГ [ГГ [Глагол *надушва*] [СГ [ЛМестоимение *аз*]]]]]

Единият вариант е да пренапишем част от граматиката по следния начин:

ЛМестоимение_н	аз   ти   той   тя  ,..
ЛМестоимение_в	мен   теб   него   нея  ,..
И	СГ_н ГГ  ,..
СГ_н	ЛМестоимение_н   СобственоИме   Съществително  ,..
СГ_в	ЛМестоимение_в   СобственоИме   Съществително  ,..
ГГ	Глагол   ГГ СГ_в  ,..
ПГ	Предлог СГ_в

По този начин се справяме с падежите, но за съжаление, и тази граматика свръхгенерира. Изречения като “Той надушва неговата вампус” или “Той надушва неговите мъртво вампус” са все още възможни, но неправилни (ако приемем, че *вампус* е в мъжки род). За да се поправи това, трябва да бъдат съгласувани рода и числото на прилагателните и глаголите със съществителните, за които се отнасят. Едната от възможностите е да се направи както при падежите на личните местоимения - тоест да се разделят на категории по род и число

прилагателните и глаголите. Това изисква да се знае родът на всички съществителни в лексикона - данни, които лесно могат да се съберат автоматично от библиотеката; нещо повече, за повечето езици може да се използва 3- или 4-грамен модел за разпознаването на рода на непознати думи.

Този подход обаче води до голямо нарастване на правилата и много пренаписване на правила. По добър подход ще бил да се разделят думите само в лексикона, а към правилата да се добавят клаузи за род, число и лице, по подобен начин на лексикализираната ВКСГ.

“лчр” - лице, число, род; “п” – падеж

Съществително_мр	вампус   Борис   стол  ,..
Съществително_ср	животно   дете   кафе  ,..
Съществително_жр	яма   Диляна   маса   птица  ,..
Съществително_мнч	вампуси   столове   животни   деца   кафета   ями  ,..
Прилагателно_мр	мъртъв   смрадлив   прав   ветровит  ,..
Прилагателно_ср	мъртво   смрадливо   право   ветровито  ,..
Прилагателно_жр	мъртва   смрадлива   права   ветровита  ,..
Прилагателно_мнч	мъртви   смрадливи   прави   ветровити  ,..
Глагол_1ЕΔ	съм  ,..
Глагол_2ЕΔ	си  ,..
Глагол_3ЕΔ	е  ,..
Глагол_1Мн	сме  ,..
Глагол_2Мн	сте  ,..
Глагол_3Мн	са  ,..
И(челна)	СГ(Номинативен, лчр, ч) ГГ(лчр, челна)  ,..
СГ(п, лчр, ч)	ЛМестоимение(п, лчр, ч)   Съществително(лчр, ч)  ,..
ГГ(лчр, челна)	ГГ(лчр, челна) СГ(Винителен, лчр_2, ч)   Глагол (лчр, челна)  ,..
ПГ(челна)	Предлог(челна) СГ(Винителен, лчр, ч)
...	...
Глагол(1ЕΔМр, ч)	Глагол_1ЕΔ<ч>
Глагол(2ЕΔМр, ч)	Глагол_2ЕΔ<ч>
...	...

Съществително(1ЕдМр,ч)	Съществително_мр<ч>
Съществително(1МнЖР,ч)	Съществително_мнч<ч>
...	...

## 18.4.9 Усложнения

Граматиката на един естествен език е много по-сложна и от последната показана граматика, която се справя с падежите, рода, лицето и числото на думите. Ще разгледаме няколко примера за сложността на естествените езици, както и евентуално начини за справяне с тях.

**Време и глаголни времена.** Трябва да може да се прави разлика между “Джон обича Мери” и “Джон обичаше Мери”. Едно възможно решение е да се добави клауза за глаголни времена. Трябва също така системата да се снабди с времева линия на събитията, за да може да различава минали от настоящи и бъдещи събития. Лесно може да се направят правила за прости сегашни, минали и бъдещи времена. Разбира се, нещата при естествените езици са много по-сложни и от това (изречението “Бил съм се бил напил” е известен пример).

**Усет за относителности.** Трябва да може да се направи връзката с неща, които не са назовани с уникалното им наименование. Например, “днес” се отнася за днешната дата, “утре вечер” се отнася за времето след 18:00 на дня след днешния, “аз” се отнася за човека, който говори; “ти” - за адресата на изречението (който може да не е системата, правеща анализа) и така нататък. Това е задача главно за интерпретирането и семантичния анализ на изречението.

**Неопределености.** Доста често изреченията могат да имат няколко смисъла, въпреки че не са винаги очевидни в нормалния говор. Изречения като “Кучето ухапа патката” може да означава и че кучето е ухапало патката, и че патката е ухапала кучето, и и двете са граматично правилни. “Надушвам вампус в 2 2” може да означава, че агента се намира в 2 2 и надушва вампус около себе си, или че агента надушва вампус, който се намира в 2 2. Грешна интерпретация в този случай може да бъде фатална.

**Метонимия.** Метонимията е изразно средство, при което един обект се назовава с чуждо име на базата на някаква връзка между тях. В ежедневната реч използваме повече метонимия, отколкото предполагаме. Когато се анализират съществителните имена, трябва да се има предвид това, ако действието изглежда невъзможно при буквalen прочит. Например, ако се каже “Интел обявиха нов чип”, става дума, че говорител на Интел е направил изявление, в което обявява новия чип. В изречението “Колата пред нас реши да завие надясно” реално шофьорът на колата е взел решението да завие.

**Пояснение.** Става дума за извлечане на най-подходящото значение според ситуацията и здравият разум. Тъй като използваме вероятностен модел, вече имаме представа за вероятностното разпределение на възможните интерпретации, но моментните обстоятелства могат да променят това. Тук трябва да се вземе под внимание това какво знаят слушателя и говорителя.

Например изречението "Аз съм мъртъв!" съдва ли означава, че говорещия е наистина мъртъв, тъй като слушателя знае, че мъртвите не могат да говорят. Далеч по-вероятно е говорителя да е имал предвид "Много съм загазил."

Това обаче е нож с две остриета и трябва да се внимава. Ето например изречението "Аз не съм кука". Наивния модел би сложил около 50% шанс за това говорителя да не е полицай, но близо 100% за това, че не е закривено парче метал - и логично би избрал втората интерпретация.

## 18.5 Заключение

Анализът на естествените езици е една от най-важните подобласти на Изкуствения Интелект. За разлика от повечето обекти на изследване от ИИ, разбирането на естествени езици налага изучаването на човешки модели на поведение, което се оказва изключително сложно, но и много интересно.

## 18.6 Речник

corpus - библиотека

smoothing - изглаждане

fall-back model - "отстъпващ" модел

regular expression / regexp - регулярен израз / регекс

tokenization - токенизация

parsing - парсване / разбор / извод

parse tree - дърво на извод

## 18.7 Използвана литература

Artificial Intelligence, A Modern Approach 3rd Edition (Russell, Norvig 2010)

Изкуствен Интелект (Нишева, Шишков 1995)

en.wikipedia.org

bg.wikipedia.org

<http://ilpubs.stanford.edu:8090/421/1/1999-65.pdf> (Extracting patterns and relations from the World Wide Web, Sergey Brin 1999)

<http://www.cs.ucdavis.edu/~rogaway/classes/120/winter12/CYK.pdf> (CYK Algorithm)

# **19 Предсказване чрез регресионни дървета.**

Регресионните дървета са структури за рекурсивно разделяне на данни на прости множества, моделирани чрез регресионни методи.

## **Съдържание**

[Съдържание](#)

[Линейна регресия](#)

[Рекурсивно разделяне](#)

[Дървета за предсказване](#)

[Рекурсивна част](#)

[Създаване на локален модел](#)

[Предимства](#)

[Пример](#)

[Разделяне на данните](#)

[Общи понятия](#)

[Алгоритъм за построяване](#)

[Проблеми](#)

[Проблеми при построяване на регресионни дървета](#)

[Хитрини и начини за отстраняване](#)

[Източници](#)

## **19.1 Линейна регресия**

Регресионните дървета са вид дървета за предсказване. В простата линейна регресия (инструмент на стратегическия анализ, който бива използван за предвиждане на бъдещите стойности на базата на съществуващи данни), реалната зависима променлива  $Y$  се моделира като линейна функция на реалната независима променлива  $X$  и шум  $\epsilon$ :

$$Y = \beta_0 + \beta_1 X + \epsilon$$

В множествената регресия имаме множество независими променливи  $X_1, X_2, \dots, X_p \equiv X$

$$Y = \beta_0 + \beta^T X + \epsilon$$

Всичко това е вярно, при положение, че всяка независима променлива има строго събирателен ефект върху  $Y$ . Възможно е обаче да съществуват и взаимодействия,

$$Y = \beta_0 + \beta^T X + \gamma X^T + \epsilon$$

Но очевидно броят на параметрите се увеличава много бързо и по-силните нелинейности ще представляват проблем.

Линейната регресия е глобален модел, където има единствена формула за предсказване над цялото пространство от данни. Когато данните имат множество сложни нелинейни взаимодействия помежду си, извеждането на единствен глобален модел може да бъде много трудно и отчайващо сложно.

### **19.1.1 Рекурсивно разделяне**

Алтернативен подход на линейната регресия е да разделяме пространството на части, така че взаимодействията между променливите да са по-прости. След това разделяме частите на още по-малки части – това се нарича рекурсивно разделяне – докато не получим толкова прости части, че да можем да създадем обикновени модели за тях. По този начин глобалният модел има две части – едината е рекурсивното разделяне, другата е създаването на прост модел за всяка клетка от разделянето.

### **19.1.2 Дървета за предсказване**

#### **Рекурсивна част**

Дърветата за предсказване използват структурата дърво за представяне на рекурсивното разделяне. Всеки от крайните върхове (или листа) на дървото, представлява клетка от разделянето и има прикрепен към него прост модел, който принадлежи единствено на тази клетка. Точката  $x$  принадлежи на листо, ако  $x$  приспада към съответната клетка за това листо. За да определим в коя клетка ще попаднем, започваме от корена на дървото и задаваме серия от въпроси и се движим по различните клони на дървото, в зависимост от дадените отговори. Вътрешните върхове на дървото са съдържат въпросите, а клоните – възможните отговори. В класическата версия всески въпрос се отнася към определено качество и има отговор с „да“ или „не“, например: „Конската сила > 50 ли е?“. Нека отбележим, че не е нужно променливите да са от един и същи тип – някои могат да бъдат безкрайни, други могат да бъдат дискретни, но подредени, трети пък могат да са по категории и т.н. Могат да се задават не само двоични въпроси, но те така или иначе винаги просто могат да се представят с по-голямо двоично дърво.

#### **Създаване на локален модел**

Толкова за рекурсивната част. Какво можем да кажем за простите локални модели? При класическите регресионни дървета, моделът във всяка клетка е просто константа стойност на  $Y$ . Да допуснем, че точките  $(x_1, y_1), (x_2, y_2), \dots, (x_c, y_c)$  са всички примери, принадлежащи на листото  $l$ . Тогава моделът за  $l$  е просто средното аритметично на зависимите променливи в клетката. Това е частично-константен модел.

#### **Предимства**

Предсказването е бързо – няма сложни сметки, а само търсене на константи в дървото

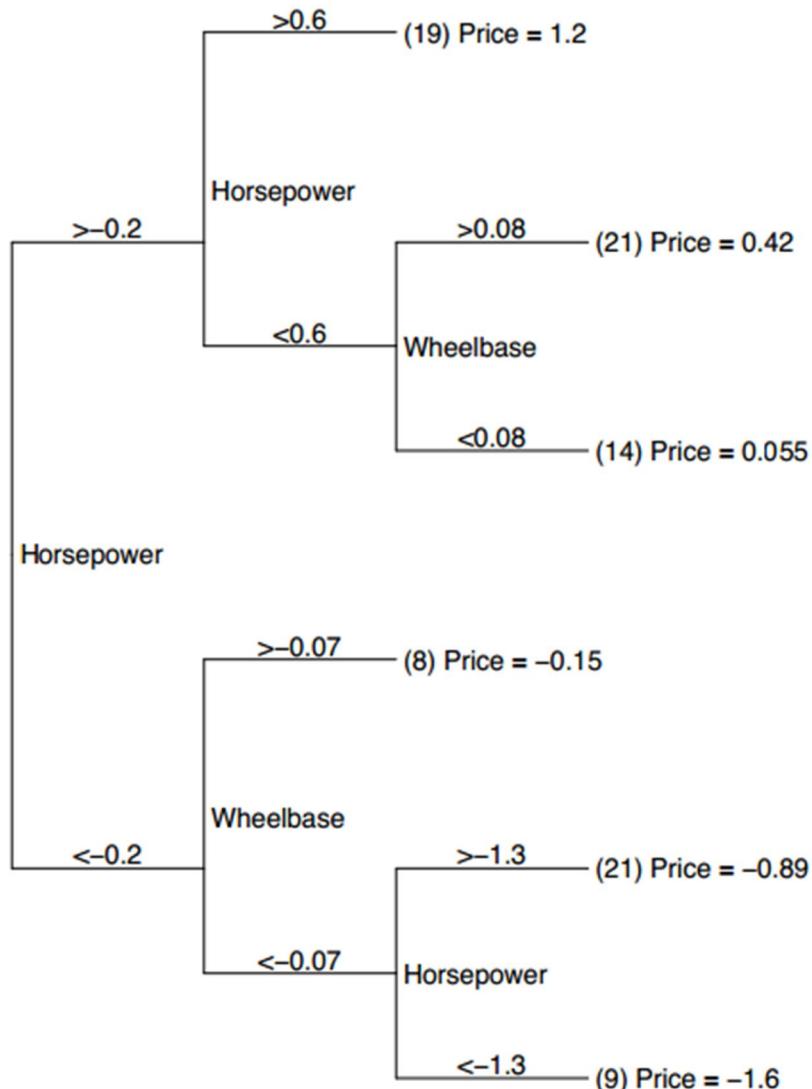
Лесно се разбира кои променливи са важни при предсказването само при поглеждане на дървото

Ако липсват данни, може да не успеем да достигне до листо на дървото, но все още можем да направим предположение като вземем средното на всички листа от поддървото, до което сме достигнали.

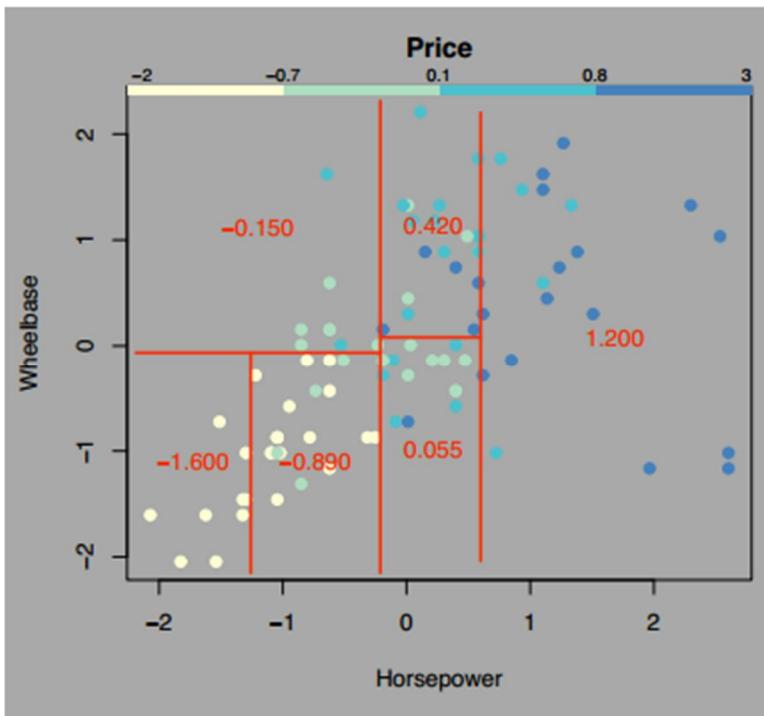
Има бързи и надеждни алгоритми за изучаване на това дърво.

### Пример

Фигура 1 е пример за регресионно дърво, което предсказва цената на коли.



Фиг 1: Регресионно дърво за предсказване цената на кола от 1993 г. Всички характеристики са стандартизириани да имат средно със стойност 0. Забелязваме, че реда, в който променливите се изследват зависи от отговорите на предходните въпроси. Числата в скоби показват колко експоната (точки) принадлежат на всяко листо.



Фиг 2: Разделянето на данните от дървото от фигура 1. Забелязваме, че всички разделящи линии са успоредни на осите, защото всеки вътрешен връх проверява дали променливата е под или над определена стойност.

Дървото коректно представя взаимодействието между Конската сила (Horsepower) и Междуосието (Wheelbase). Когато конската сила е  $> 0.6$ , междуосието няма значение. Когато двете са еднакво важни, дървото се разклонява помежду им.

Веднъж построим ли дървото, локалните модели стават напълно определени и лесни за намиране (просто апроксимираме), така че всичките усилия се насочват единствено към намирането на добро дърво, иначе казано – намиране на добър начин за разделяне на данните. Можем да вземем добри идеи за това от клъстеризицата.

## 19.2 Разделяне на данните

### Общи понятия

При регресионните дървета искаме да максимизираме  $I[C; Y]$ , където  $Y$  е зависимата променлива, а  $C$  е променливата, която индицира на кое листо от дървото завършваме. Тъй като не можем да извършим директна максимизация, прилагаме greedy search. Започваме с намиране на бинарен въпрос, който максимизира информацията, която имаме за  $Y$ ; това ни дава корена на дървото и двата му дъщерни върха. За всеки от дъщерните върхове, повтаряме процедурата за корена, питайки се кой въпрос би ни дал максимална информация за  $Y$ . И така повтаряме процедурата рекурсивно.

Също както при клъстеризицата обаче, бихме могли да достигнем до поставянето на всяка точка във свое собствено листо, което няма да ни бъде от голяма помощ. Един от типичните критерии за спиране разклоняването на дървото е когато по-нататъчно разклоняване би дало

минимална нова информация или пък когато новите върхове биха съдържали например по-малко от 5% от всичките данни.

Сумата от квадратните грешки за дърво T е:

$$S = \sum_{c \in \text{leaves}(T)} \sum_{i \in C} (y_i - m_c)^2$$

, където

$$m_c = \frac{1}{n_c} \sum_{i \in C} y_i$$

е предсказването за листото с. Това може да бъде пренаписано като:

$$S = \sum_{c \in \text{leaves}(T)} n_c V_c$$

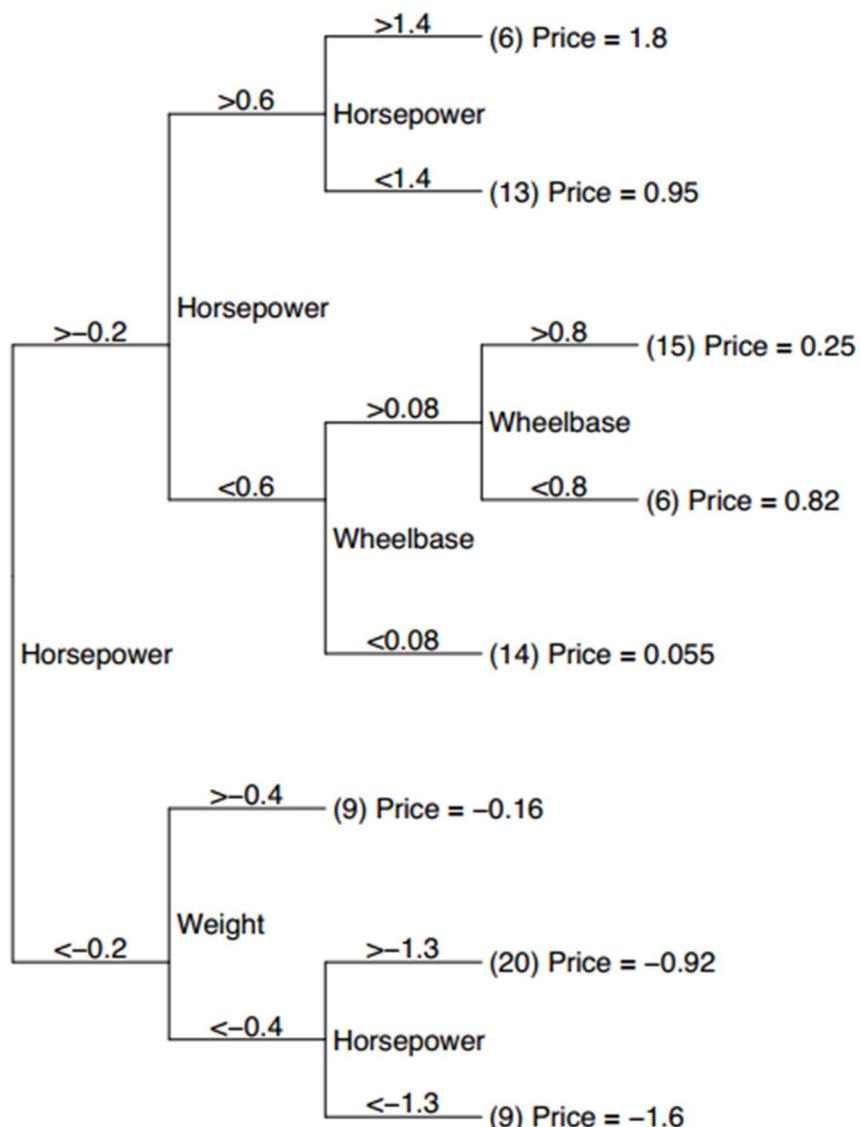
където  $V_c$  е дисперсията за листото с. Така ще направим разклоняването минимизирайки S.

### **Алгоритъм за построяване**

Основният алгоритъм за построяване на регресионно дърво е:

1. Започваме с един връх, който съдържа всички точки. Смятаме тс и S.
2. Ако всички точки от върха имат една и съща стойност за всички независими променливи, спираме. Иначе, търсим по всички двоични разклонения на всички променливи за тази, която редуцира S, колкото се може повече. Ако най-малката стойност за S е по-малка от някакъв праг  $\delta$ , или някой от новите върхове би съдържал по-малко от q точки, спираме. Иначе, взимаме разклонението и създаваме два нови върха.
3. За всеки от новите върхове се връщаме към стъпка 1.

Дърветата използват само една независима променлива на всяка стъпка. Ако няколко променливи са еднакво добри, няма значение коя се използва (Примера на фиг. 3 показва, че тежестта (Weight) е точно толкова добра колкото междуосието (Wheelbase))



Фиг 3: Друг пример за регресионно дърво за цената на коли, където тежестта (Weight) е използвана вместо междуосните (Wheelbase). Двете дървета са еднакво бързи.

### 19.3 Проблеми

#### Проблеми при построяването на регресионни дървета

Един проблем с предходния алгоритъм за построяване на дървото е, че може да спре прекалено рано. Възможно е да има променливи, които не носят много информация сами за себе си, но водят до високо информативни разклонения след това. Такъв проблем може да се появи, когато  $S$  стане по-малко от някаква  $\delta$  или пък от произволно избиране на минимален брой точки  $q$  за връх.

#### Хитрини и начини за отстраняване

Един по-успешен подход за построяване на регресионно дърво е крос-валидацията. На случаен принцип разделяме данните си на 2 части. След това прилагаме основния алгоритъм

на едната част с  $q = 1$  и  $\delta = 0$  – по този начин построяваме възможно най-голямото дърво. Това дърво ще бъде твърде тежко и препълнено с данни, така че използваме крос-валидация за отсичане клоните на дървото. За всяка двойка листа с общ баща, оценяваме грешките чрез другата част от данните и виждаме дали сумата от квадратите би била по-малка, ако махнем тези 2 листа и направим техния баща листо. Това се повтаря, докато отсичането спре да намалява грешките.

Има много други хитрини при построяване на дървета. Интересен такъв е да комбинираме растегнето и отсичането като ги редуваме. Разделяме данните отново на 2 части и първо построяваме дървото от едната част и отсичаме излишните клони. След това разменяме двете части и започваме да отсичаме другата част. След това отново разменяме и отсичаме първата част. Повтаряме това докато дървото не спре да променя размера си.

## **19.4 Източници**

Artificial Intelligence, A Modern Approach - 2nd Edition

<http://www.stat.cmu.edu/~cshalizi/350-2006/lecture-10.pdf>

[http://en.wiktionary.org/wiki/regression\\_tree](http://en.wiktionary.org/wiki/regression_tree)

# **20 Извличане на асоциативни правила**

## **Association Rules Mining**

### **Съдържание**

1. Въведение
2. Основни понятия
3. Основни алгоритми за извлечане на асоциативни правила
4. Повишаване на ефективността на алгоритмите
  - a. Намаляване на броя минавания през базата данни
  - b. Взимане на извадки
5. Литература

### **20.1 Въведение**

Извличане на асоциативни правила е една от най-важните и добре проучени техники на извлечане на данни. Тя има за цел да извлече интересни корелации, чести модели, асоциации или случайни структури сред набор от елементи в транзакционни бази данни или други хранилища за данни. Асоциативните правила са широко използвани в различни области като телекомуникационни мрежи, пазар и управление на риска, инвентаризационен контрол и др.

Извличането на асоциативни правила има за цел да разберете за асоциативните правила, които отговарят на предварително предварително зададена минимална подкрепа и доверие от дадена база данни. Проблемът обикновено се разгражда на два подпроблема. Един от тях е да се намерят тези множества от предмети, чиито появявания надвишава предварително определения праг в базата данни; тези множества се наричат "чести" или "големи". Вторият проблем е да се генерират асоциативни правила от тези големи множества с ограниченията на минимална подкрепа. Да предположим, че един от големите множества е  $L_k$ ,  $L_k = \{I_1, I_2, \dots, I_k\}$  асоциативни правила с тези множества се генерират по следния начин: първото правило е  $\{I_1, I_2, \dots, I_{k-1}\} \Rightarrow \{I_k\}$ , като проверим доверието на това правило можем да го определим като интересно или не. Тъй като вторият подпроблем е доста ясен, по-голямата част от изследванията се фокусират на първия подпроблем.

В много случаи, алгоритмите генерираат на изключително голям брой асоциативни правила, често хиляди или дори милиони. Освен това, правилата за асоцииране понякога са твърде големи. Почти невъзможно е за крайните потребители да разберат или валидира такъв голям брой сложни правила за асоцииране, като по този начин се ограничава полезнотата на резултата от извлечането на данни. Били са предложени няколко стратегии за намаляване на броя на асоциативните правила, като генериране само на "интересни" правила, генериране само на "не излишни" правила, или, генериране само на тези правила, които отговарят на определени други критерии.

## **20.2 Основни понятия**

Нека  $I = I_1, I_2, \dots, I_m$  набор от  $m$  различни атрибути,  $T$  е транзакция, която съдържа набор елементи, така че  $T \subseteq I$ ,  $D$  база данни с различна транзакционни записи  $T_s$ . Едно асоциативно правило е предположение под формата на  $X \Rightarrow Y$ , където  $X, Y \subset I$  са набори от елементи, наречен множества и  $X \cap Y = \emptyset$ .  $X$  се нарича предшестващ, а  $Y$  се нарича следствие.

Правило означава,  $X$  предполага  $Y$ .

Има два важни основни мерки за асоциативните правила, подкрепа (и) и доверие (в). Тъй като базата данни е голяма и потребителите се интересуват само от онези често закупувани предмети, то обикновено са предварително определени прагове на подкрепа и доверие от страна на потребителите да се отхвърлят тези правила, които не са толкова интересни или полезни. Двета прага се наричат минимална подкрепа и минимално доверие. Support (ите) на асоциативното правило се определя като процент / част от записите, които съдържат  $X \cup Y$  към общия брой записи в базата данни. Да предположим, че подкрепата на елемент е 0.1%, това означава, едва 0,1% от транзакцията съдържат закупуването на този продукт.

Доверието на правило за асоцииране се определя като процент / част от броя на сделки, които съдържат  $X \cup Y$  към общия брой на записи, които съдържат  $X$ . Доверието е мярка за силата на асоциативните правила. Да предположим, че доверието на асоциативното правило  $X \Rightarrow Y$  е 80%, това означава, че 80% от транзакциите, които съдържат  $X$  съдържат и  $Y$  заедно.

Като цяло, набор от елементи (като предхода или следствието на правило) се нарича покупки. Броят на елементите в едно такова множество се нарича дължината на покупките.

Покупки с някаква дължина  $K$  се наричат  $K$ -покупки.

## **20.3 Основни алгоритми за извлечение на асоциативни правила**

AIS алгоритъм е първият алгоритъм, предложен за извлечение на асоциативни правила.

В този алгоритъм се генерираат асоциативни правила със следствие само с един елемент, което означава, че следствието на тези правила съдържа само един елемент, например ние само генерираме правила като  $X \cap Y \Rightarrow Z$ , но не правила като  $X \Rightarrow Y \cap Z$ . Основният недостатък на AIS алгоритъма е твърде много покупки кандидатки, които най-накрая се оказат малки са генериирани, което изисква повече пространство и хаби много усилия, които се оказат безполезни. В същото време този алгоритъм изисква твърде много пасове през цялата база данни.

Apriori е по-ефективно по време на процеса на генериране на кандидат. Apriori използва съкращаващи техники, за да се избегне измерването на някои покупки, като същевременно се гарантира пълнотата. Такива покупки, алгоритъмът може да докаже, няма да се окажат големи. Все пак има две пречки на алгоритъма Apriori. Един от тях е сложният процес за генериране на кандидат, който използва повече време, пространство и памет.

Друга пречка е многократно сканиране на базата данни. Въз основа на алгоритъм Apriori, много нови алгоритми са проектирани с някои изменения или подобрения.

## **20.4 Повишаване на ефективността на алгоритмите**

Изчислителната стойност на извлечането на асоциативно правило може да бъде намалена по четири начина:

- чрез намаляване на броя на минавания/пасове през базата данни
- извадка от базата данни
- чрез добавяне на допълнителни ограничения за структурата на моделите
- чрез паралелизация.

През последните години е постигнат голям напредък във всички тези посоки.

### **20.4.1 Намаляване на броя минавания през базата данни**

FP-Tree, чест модел за извлечане, е друга важна стъпка в развитието на извлечането на асоциативните правила, който разгражда основните пречки на Apriori. Честите покупки се генерираят само с две минавания през базата данни и без никакъв процес за генериране на кандидат. FP-дърво е продължителен префикс дърворидна структура съхраняваща важна, количествена информация за често срещаните модели. Само "чести" съдължина 1 обекти ще имат възли в дървото, и възлите на дървото са разположени по такъв начин, че по-често срещаните възли ще има по-добри шансове за споделяне на възли, отколкото по-рядко настъпили такива.

FP-Tree скапира много по-добре, отколкото Apriori, заплото като прага подкрепата намалява, както и дължината на чести покупки се увеличава драстично.

Множеството от кандидати, с които Apriori трябва да се справи е изключително голям, и моделът на свързването на кандидати от търсене чрез сделките става много скъп. Процесът за генериране на чести модели включва два под процеса:

- -Изграждане на FP-Tree
- -Генериране на чести модели от FP-Tree.

Резултатът от извлечането е същият с Apriori серия алгоритми. За да обобщим, ефективността на FP-Tree алгоритъм се забелязва в три точки.

- Първо: FP-Tree е компресирано представяне на оригиналната база данни, заплото само тези, "чести" елементи се използват за изграждане на дървото, друга ненужна информация се изрязва.
- На второ място този алгоритъм сканира базата данни само два пъти.
- На трето място, FP-Tree използва разделен и владей метод, който значително намалява размера на последващото условно FP-Tree.

Всеки алгоритъм има своите ограничения, за FP-Tree е трудно да бъде използван в интерактивна система за извлечане на данни. По време на интерактивен процес на извлечане, потребителите могат да променят прага на подкрепа, в съответствие с правилата. Въпреки това, за FP-Tree промяна на подкрепата може да доведе до повторение на целия процес на

извличане. Друго ограничение е, че FP-Tree не е подходящ за инкрементално извличане. Тъй като с течение на времето базите данни се променят непрекъснато, нови бази данни може да се добавят към тях т.н. Тези вмъквания могат да доведат до повторение на целия процес, ако използваме FP-Tree алгоритъм.

TreeProjection е друг ефективен алгоритъм. Общата идеята на TreeProjection е, че тя изгражда лексикографско дърво и проектира голяма база данни върху множество от намалени, базирани на предмети под-бази базирани на честите модели извлечени досега. Броят на възлите в лексикографското дърво е точно толкова, колкото са и честите покупки. Ефективността на TreeProjection може да се обясни с два основни фактора:

- (1) проекцията на транзакцията ограничава броя на подкрепата в едно сравнително малко пространство, и
- (2) лексикографското дърво улеснява управлението и броенето на кандидатите и осигурява гъвкавостта на избор на ефективна стратегия по време на генерирането на дървото и проекцията на транзакцията.

Уанг и Тјортис представят PRICES, ефективен алгоритъм за извличане на асоциативни правила. Техният подход намалява времето стъпката за генериране на покупки, известна с това, че отнема най-много време, чрез сканиране на базата данни само веднъж и използването на логически операции в процеса.

Друг алгоритъм за ефективно генериране на големи чести множества кандидати се нарича Matrix алгоритъм. Алгоритъмът генерира матрица, в която се вписват 1 или 0, като минава през базата данни само веднъж, и след това честите множества кандидати се получават от матрицата. Накрая асоциативните правила се извличат от честите множества кандидати.

Експерименти резултати потвърждават, че предложеният алгоритъм е по-ефективен от Apriori алгоритъма.

#### **20.4.2 Вземане на извадки**

Toivonen, представен като алгоритъм за извличане на асоциативни правила чрез вземане на извадки. Подходът може да бъде разделен на две фази. По време на фаза 1 извадка на базата данни е получена и всички асоциации в извадката са открити. Тези резултати са валидирани от цялата база данни. За да увеличи ефективността на генералния подход авторът се възползва от занизената минимална подкрепа на извадката. Тъй като този подход е пробабилистичен (т.е. зависи от пробата съдържаща всички важни асоциации) не всички асоциативни правила могат да бъдат извлечени на този първи пас. Тези асоциации които са "не чести" във извадката, но всъщност са "чести" във цялата база данни се използват за да допълнят множеството от асоциации във фаза 2.

Chuang и др. разглеждат друг прогресивен алгоритъм за вземане на извадки, наречен Sampling Error Estimation(SEE), който цели да идентифицира подходящ размер на извадка за извличане на асоциативни правила. SEE има две приимущество. Първо: SEE е много ефективен защото подходяща големина на извадката може да бъде определен без да е необходимо да се изпълняват асоциативни правила. Второ: идентифицираният размер на извадка от SEE е много точен, което означава че асоциативни правила могат да бъдат

изключително ефективни на извадка от този размер за да се постигнат достатъчно точен резултат.

Особено ако данните идват като поток, преминаващи в по-бързи темпове, отколкото могат да се преработят -то взимането на извадки изглежда като единственото решение

#### **20.4.3 Паралелизация**

Техники за извлечане на асоциативни правила постепенно се адаптират към паралелни системи с цел да се възползват от по-бързите скорости и по-големия капацитет за съхранение които се предлагат. Преходът към разпределена система изисква разделяне на базата данни сред процесорте - процедура, която обикновено се извършва безразборно.

Cheung и др представят алгоритъм, наречен FDM. FDM е паралелизация на Apriori (свързани с нищо машини, всеки със своя собствен дял на базата данни. На всяко ниво и на всяка машина, сканиране на базата данни се извършва независимо от местния партньор. Тогава разпределена съкращаваща техника се използва.

Шустер и Wolff описват друг Apriori D-ARM алгоритъм - DDM. Както в FDM, кандидатите в DDM се генерираят levelwise и след това се отчитат от всеки възел в своята локална база данни. Възлите изпълняват разпределено протоколно решение, за да открият кои от кандидатите са чести и кои не са.

### **20.5 Литература**

[http://en.wikipedia.org/wiki/Association\\_rule\\_learning](http://en.wikipedia.org/wiki/Association_rule_learning)

Agrawal, R., Imielinski, T., and Swami, A. N. 1993. Mining association rules between sets of items in large databases.

# 21 Метод за клъстеризиация DBSCAN

## *Density-Based Spatial Clustering of Application with Noise*

### **Съдържание**

Въведение

Формални дефиниции

Коментар на дефинициите

Алгоритъм

Псевдокод

Оценка на сложността

Параметрите Eps и MinPts

Предимства на алгоритъма

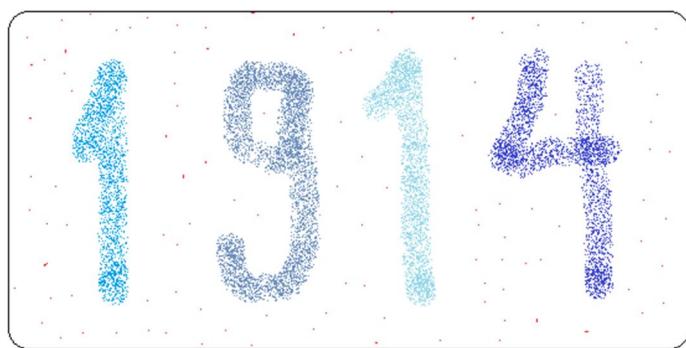
Недостатъци

Речник

Литература

### **21.1 Въведение**

Ако разглеждаме едно множество от точки в пространството, можем лесно да различим отделните сегменти (клъстери), както и шумови (не принадлежащи на нито един клъстър) точки.



Това е така, защото плътността на точките във вътрешността на всеки сегмент е осезаемо по-голяма от тази на точките извън него. Методът DBSCAN прави плътностно базирано клъстериране на пространствено множество данни, в което има и фонов шум. Такова нещо би ни се наложило да правим, ако имаме за задача да извлечем знания от снимкова база от данни, автоматично събрана от сателити или пътни камери. Една предварителна обработка на снимките, включваща намирането на съставните части (къщи, пътища, реки, автомобили, номер на автомобил, и т.н.), би ускорила и улеснила по-нататъшната ни работа. Такова отделяне се постига чрез алгоритмите за клъстеризиация.

При алгоритмите за кълстеризация, често възникват няколко проблема. Има случаи, когато е трудно да се съберат предварителни знания за конкретните данни (примерно, брой кълстери). Също така пространствените бази данни са доста обемни и когато имат няколко измерения, обработката им ще изиска голяма изчислителна мощ. Друг проблем е, че геометричната форма на кълстера може да е доста сложна. Алгоритми като CLARANS, K-means, K-medoid, Hierarchical Clustering and Self-Organized Maps, за жалост не могат да се справят добре с посочените проблеми.

Но за всеобща радост, предложението през 1996 от Martin Ester, Hans-Peter Kriegel, Jörg Sander and Xiaowei Xu [2] и разгледан в настоящата тема, алгоритъмът Density-Based Spatial Clustering of Application with Noise, успешно разрешава и трите проблема.

## 21.2 Формални дефиниции

**Дефиниция 0 (за дефиниционната област):** това са всички разглеждани точки. Означение:  $D$ .

**Дефиниция 1 (за епсилон-съседите):** Епсилон-съседи на дадена точка  $p$ , ще наричаме всички точки от  $D$ , които попадат в епсилон-околността на  $p$ . Означение:  $N_{Eps}(p)$

**Дефиниция 2 (за директно плътностно-достижимите точки):** Ще назоваме, че точката  $q$  е директно плътностно-достижима от точката  $p$ , относно параметрите  $Eps$  и  $MinPts$ , ако:

- 1)  $p$  е от  $N_{Eps}(q)$
- 2)  $|N_{Eps}(q)| \geq MinPts$

**Дефиниция 3 (за плътностно-достижимите точки):** Ще назоваме, че точката  $q$  е плътностно-достижима от точката  $p$ , относно параметрите  $Eps$  и  $MinPts$ , ако съществува редица от точки  $p_1, p_2, \dots, p_n$ ,  $p_1 = q, p_n = p$ , такава че  $p_{i+1}$  е директно плътностно-достижима от  $p_i$ .

**Дефиниция 4 (за плътностна-свързаност):** Ще назоваме, че точката  $p$  е плътностно-свързана с точката  $q$ , относно параметрите  $Eps$  и  $MinPts$ , ако съществува точка  $t$ , такава че двойките  $p, t$  и  $q, t$  са плътностно-свързани, относно параметрите  $Eps$  и  $MinPts$ .

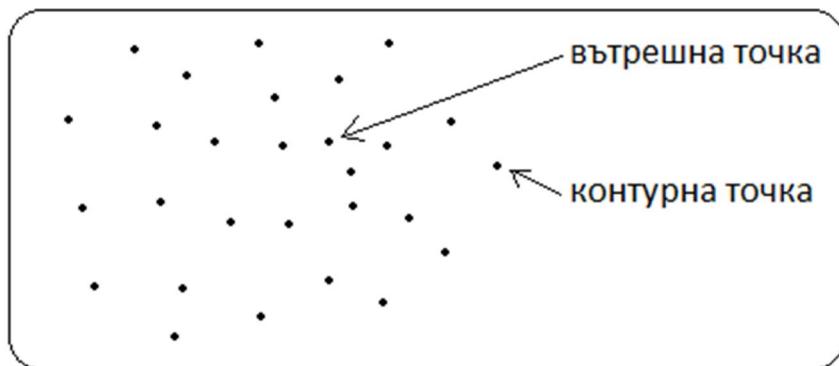
**Дефиниция 5 (за кълстер):** Ще назоваме, че непразното подмножеството точки  $C$  на множеството  $D$ , е кълстер на  $D$ , относно параметрите  $Eps$  и  $MinPts$ , ако:

- 1) За всеки точки  $p$  и  $q$ , ако  $p$  е от  $C$  и  $q$  е плътностно-достижима от  $p$ , относно параметрите  $Eps$  и  $MinPts$ , то и  $q$  е от  $C$
- 2) Всеки две точки от  $C$  са плътностно-свързани, относно параметрите  $Eps$  и  $MinPts$ .

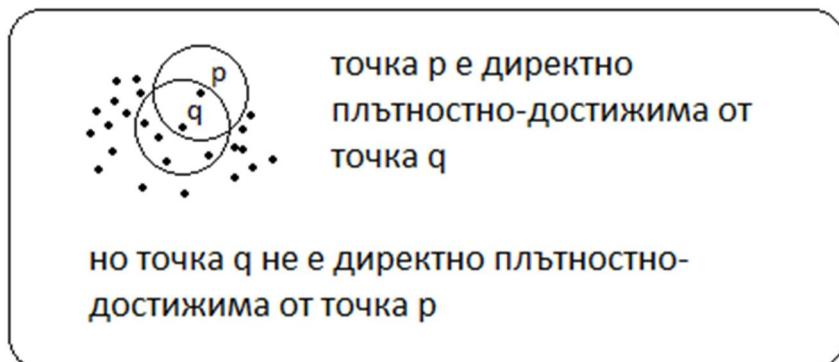
**Дефиниция 6 (за шум):** Нека  $C_1, C_2, \dots, C_k$  са кълстери на  $D$ , относно параметрите  $Eps_i$  и  $MinPts_i$ ,  $i=1,\dots,k$ . Тогава шум ще наричаме множеството от точки на  $D$ , които не попадат в нито един от кълстерите  $C_i$ .

### 21.3 Коментар на дефинициите

Логично е да си помислим, че всяка точка в даден кълстер има поне MinPts на брой точки в своя списък с епсилон-съседи. Но това се оказва грешно. В даден кълстер има два вида точки. Едните имат поне MinPts съседа в епсилон околността си. Тези точки ще наричаме вътрешни точки. Другият вид са точките от контура на кълстера, те съдържат по-малък брой съседи, но продължават да са точки от същия кълстер. За краткост ще ги наричаме контурни точки.



Втората дефиниция е симетрична само, ако става дума за вътрешни точки. Тя очевидно не е симетрична, ако се прилага върху вътрешна и контурна точка.



Дефиницията 3, за плътностно-достижимите точки, е разширение на предходната, транзитивна е, но отново е не симетрична.



Непът повече две контурни точки на същия кълстер, може да се окажат не плътностно-достижими една от друга.



Дефиницията за плътностно-свързаните точки вече е симетрична.



Всеки множество, което е кълстер C, относно параметрите Eps и MinPts, съдържа поне MinPts на брой точки.

## 21.4 Алгоритъм

DBSCAN алгоритъмът изисква два параметъра: Eps и MinPts. Всички намери от него кълстери са относно тези параметри.

За да намерим кълстер избираме коя да е неизползвана точка р. Намираме всички точки, които са плътностно-достижими от нея, относно параметрите Eps и MinPts. Ако броятката е повече от MinPts създаваме нов кълстер и отбелязваме всички точки като използвани. Ако не са били достатъчен брой отбелязваме р като използвана и я прибаваме към списъка с шумовите точки. Продължаваме да търсим кълстери докато имаме неизползвани точки. (Забележка: точка, отбелязана като шумова, може на по-следваща итерация да се окаже част от ладен кълстер. В такъв случай трябва да се извади от списъка на шумовите.)

### 21.4.1 Псевдокод (източник: [1])

```

DBSCAN(D, eps, MinPts)
C = 0
for each unvisited point P in dataset D
    mark P as visited
    NeighborPts = regionQuery(P, eps)
    if |NeighborPts| < MinPts
        C += 1
        for each point Q in NeighborPts
            if Q is unvisited
                mark Q as visited
                NeighborPtsQ = regionQuery(Q, eps)
                if |NeighborPtsQ| >= MinPts
                    for each point R in NeighborPtsQ
                        if R is unvisited
                            mark R as visited
                            NeighborPtsR = regionQuery(R, eps)
                            if |NeighborPtsR| >= MinPts
                                for each point S in NeighborPtsR
                                    if S is unvisited
                                        mark S as visited

```

```

if sizeof(NeighborPts) < MinPts
    mark P as NOISE
else
    C = next cluster
    expandCluster(P, NeighborPts, C, eps, MinPts)

expandCluster(P, NeighborPts, C, eps, MinPts)
add P to cluster C
for each point P' in NeighborPts
    if P' is not visited
        mark P' as visited
    NeighborPts' = regionQuery(P', eps)
    if sizeof(NeighborPts') >= MinPts
        NeighborPts = NeighborPts joined with NeighborPts'
        if P' is not yet member of any cluster
            add P' to cluster C
regionQuery(P, eps)
return all points within P's eps-neighborhood

```

## 21.4.2 Оценка на сложността

За всяка точка `regionQuery` се извиква точно по веднъж и сложността на алгоритъма зависи основно от неговата сложност. Ако се използва подходяща структура от данни това извикване може да е от порядъка на  $O(\log N)$ , което води до  $O(N \log N)$  сложност на алгоритъма по време. Ако се пази матрица с разстоянията между всеки две точки ще имаме  $O(N^2)$  сложност по памет.

## 21.4.3 Параметрите Eps и MinPts

Методът DBSCAN не остава по-назад от другите задачи за извличане на знания от данни и също като тях придава доста голямо значение на избора на параметрите си. В конкретния случай това са двата параметъра `MinPts` и `Eps`. Един неподходящ тихен избор може да доведе до неоптимално решение и до сливане на два кълстера с различна плътност.

Тези параметри могат да се определят чрез прости и ефективна евристика ([глава 4.2 на \[2\]](#)):

Наблюдение: Нека  $d$  е разстоянието от точка  $p$  до най-близък  $k$ -ти съсед. Тогава в  $d$ -околността на точка  $p$  ще има точно  $k+1$  за почти всяка точка  $p$ . ( $d$ -околността на точка  $p$  ще има повече от  $k+1$  точки, само тогава когато от  $p$  има поне две точки на разстояние  $d$ .) Ако променяме  $k$  за дадена точка  $p$ , това в общия случай няма да води до драстични промени на  $d$ .

(Сериозни промени в разстоянието  $d$  ще настъпват, само ако  $k$ -тите най-близки съседи на  $p$  са разположени на една права. Но в общия случай това не е вярно за точките на даден кълстер.)

Конструкция: Нека дефинираме функцията  $k\text{-dist}$  съпоставяща на всяка точка  $p$ , разстоянието  $d$  до нейния най-близък  $k$ -ти съсед. Нека сортираме точките по тяхната  $k\text{-dist}$  стойност в намаляващ ред. Нека означим графиката на сортирани  $k\text{-dist}$  стойности като  $\text{sorted } k\text{-dist}$ .

Сега, нека да фиксираме някоя точка  $p$ . Нека за параметрите  $Eps$  и  $\text{MinPts}$  изберем стойностите  $k\text{-dist}(p)$  и  $k$ , съответно. Тогава всички точки с по-малка  $k\text{-dist}$  стойност, ще бъдат вътрешни точки за кълстерите.

Ако намерим точка  $t$ , такава че тя да е точката от „най-тънкия“ кълстер с максимална  $k\text{-dist}$  стойност в него, то тогава ще сме намерили желаните стойности и за параметрите на алгоритъма. Такава е точка е първата точка на първото „равно“ място на графиката  $\text{sorted } k\text{-dist}$ . Всички точки вляво от нея ще са шумови, а останалите ще принадлежат на някой кълстер.



Според [2]: при  $k>4$ , графиките  $\text{sorted } k\text{-dist}$  не се различават съществено от графиката  $\text{sorted } 4\text{-dist}$  за двумерни данни. Това ни дава възможността (за двумерни данни) да фиксираме параметъра  $\text{MinPts}$  на 4. Сега вече можем след като изчислим и визуализираме графиката  $\text{sorted } 4\text{-dist}$  ръчно да изберем точката  $t$  и за  $Eps$  да изберем  $4\text{-dist}(t)$ .

Още по темата за параметрите може да бъде намерено в [3].

#### 21.4.4 Предимства на алгоритъма

DBSCAN не изисква предварително знание за броя на кълстерите.

DBSCAN намира кълстери с всякакъв контур. Включителни и такива, които са изцяло заобиколени от друг кълстер.

DBSCAN взима предвид наличието на шум и го игнорира.

DBSCAN изисква само два параметъра  $Eps$  и  $\text{MinPts}$  и е почти независим от подредбата на точките. (Но все пак е възможно, подредбата да разменят контурни точки на два кълстера)

#### 21.4.5 Недостатъци

Както и при всички останали алгоритми, качеството на DBSCAN зависи от избора на функцията, определяща разстоянието между точките.

DBSCAN не може да кълстерира добре множества, в което има голямо разминаване между различните плътности.

## 21.5 Речник

cluster	кълстер
density-based	плътностно базирано
spatial	пространствено
noise	фонов шум
epsilon-neighborhood	епсилон-съседи
core points	вътрешни точки
border points	контурни точки
noise points	шумови точки
directly density-reachable	директно плътностно-достижим
density-reachable	плътностно-достижим
density-connected	плътностно-свързан

## 21.6 Литература

[1]: <http://en.wikipedia.org/wiki/DBSCAN>

[2]: "*A density-based algorithm for discovering clusters in large spatial databases with noise*". Martin Ester, [Hans-Peter Kriegel](#), Jörg Sander, Xiaowei Xu (1996-)

[3]: Consistency and rates for clustering with DBSCAN:  
<http://jmlr.csail.mit.edu/proceedings/papers/v22/sriperumbudur12/sriperumbudur12.pdf>

## **22 Намаляване на обема на данни (компресия)**

Компресията на данни е изкуството на намаляване на броя нужни битове за представяне и съхранение на данни.

### **Съдържание**

[Съдържание](#)

[Компресиране на данни](#)

[Методи за компресия без загуба](#)

[Методи за компресия със загуба](#)

[Резюме](#)

[Ресурси](#)

### **22.1 Компресиране на данни**

Компресирането на данни е клон в математиката, който води началото си от втората половина на 40-те години. За първоизточник може да се смята Клод Шенън, част от Bell Labs, който разглежда различни въпроси свързани със съхраняването и предаването на информация. Компресирането е част от Теорията на информацията, поради факта, че се занимава с излишеството на информация в дадено съобщение. Излишната информация в един екземпляр от даден набор данни е причина да се използват повече битове за кодирането му. Основна идея на кодирането е премахването на излишествата с цел намаляване на размера на съобщенията. Най общо казано компресирането на данни е превръщане на потока от входни данни в кодове. Ако то е ефективно, полученият поток е по-кратък от входящия поток данни, т.е. имаме намаляване на размера. Съпоставянето на определен знак или поредица от знаци с определен код се базира на даден модел. Тоест компресирането може да се разгледа като моделиране плюс кодиране.

#### **22.1.1 Ентропия**

В теорията на информацията се използва термина ентропия, като мярка колко информация е закодирана в дадено съобщение. Колкото по-висока е ентропията, толкова по-голямо е информационното му съдържание. Ентропията на отделен символ се дефинира като отрицателен логаритъм при основа две, а ентропията на цяло съобщение е сумата от ентропиите на всички символи в съобщението.

#### **22.1.2 Ниво на компресия**

За един алгоритъм е важно какво ниво на компресия постига той. Алгоритмите се сравняват в три категории:

- Време за компресиране на даден файл.
- Ниво на намаляване на размера.

- Ниво на загуба на качеството при компресирането на данните.

За изчисляването на нивото на компресия се използват различни формули, но своеобразен стандарт е станала формулата (компресиран размер / некомпресиран размер) \* 100, и се измерва в проценти. Файл, който е намалил обема си два пъти ще е с ниво на компресията 50%, а ако размера на даден файл не се промени след компресията – 100%. Този метод за измерване не е перфектен, но показва относително недвусмислено нивото на компресия на даден файл. Съществуват много други аспекти на компресирането като сложност на Колмогоров, overhead, redundancy и др., но изброените по-горе са сред най-важните.

### **22.1.3 Загуба на информацията**

Методите за компресия на данните се делят най-общо на две основни области – компресия със (фиг. 1.4) и компресия без загуба (фиг. 1.3) на информацията. При компресирането със загуба на информацията, както името подсказва, се губи част от информацията за сметка на многократно увеличаване на компресията. То е изключително широко прилагано в областта на видео и аудио обработката. Повечето от методите за такава компресия могат да се настройват на различни нива на качество, което естествено променя и степента на компресия. Избора на настройки зависи изцяло от предназначението на конкретната информация.

Методите за компресиране без загуба на информацията са тези, които гарантират, че при цикъла компресиране-декомпресиране изходът ще съвпада с входните данни. Тези алгоритми са най-често използваните за архивиране на програмен код, бази данни, електронни таблици, текстови документи, тъй като при подобни данни дори само загубата на един бит може да произведе крайно различен резултат.

### **22.1.4 Модел на компресия**

Моделът е набор от правила как да се обработват и съпоставят определени кодове на различните знаци или поредици от знаци. Компресиращият алгоритъм / програма използва модела за да определи вероятността на всеки символ, за да може кодиращият алгоритъм да получи съответния код, базиран на тези вероятности. Един от първите статистически модели за кодиране е разработен от Хъфман и става основа за всички следващи разработки на статистически модели за кодиране.

### **22.1.5 Значение за изкуствения интелект**

Очевидното приложение на компресията при работата с изкуствения интелект е възможността за намаляне на обема на данни или, другояче казано – знания. По-интересен е паралелът, който много от работещите в сферата на изкуствения интелект и информатиката като цяло, правят между компресията и изкуствения интелект в най-генералното значение на понятието. Популярна е теорията, че компресирането включващо предвиждане е подобно на изкуствен интелект, който се опитва да „чете“ естествен език, т.е. ако може да се постигне „идеална“ компресия на дадена информация, то това би било алгоритъм, способен да реши  $p(A|Q) = p(QA)/P(Q)$ , където Q е контекста до момента на задаване на A, което е еквивалентно именно на изкуствен интелект, способен да прави толкова точни „предвиждания“, или оценки, че да генерира отговори, неразличими от тези на реален човек.

## **22.2 Методи за компресия без загуба**

Съществуват многообразни методи за компресия без загуба, повечето от които са извън обхвата на тази курсова работа, било то поради прекалената си сложност или не особено широката си популярност. Има, обаче, няколко по-популярни и исторически значими методи, първия широко известен и ефективен от които е метода на Шанон-Фано.

### **22.2.1 Метод на Шанон-Фано**

К. Шанон (Шенън) от Bell Labs и M. Fano от М.I.T. Предлагат почти едновременно метод, който изисква само да е известна вероятността за срещането на всеки символ. Бивайки известна тази вероятност, може да се състави таблица, за която важат следните особености:

- Различните кодове имат различен брой битове.
- Код - на символ с по-ниска вероятност има повече битове от тези с по-висока
- Кодовете могат да бъдат декодирани еднозначно (въпреки разликите в дълчините им)

Създаването на код с дължина, зависеща от вероятността на символите, прави възможно компресирането на информацията. Проблема с еднозначното кодиране бива решен от подреждането им в двоично дърво. Дървото е и в основата на метода. То се построява по специфичен алгоритъм (забележка: алгоритъма е резюмиран, тъй като метода се разгледа в детайли, би трябвало да му се посвети отделна курсова работа):

За всяка поредица от символи се създава списък от вероятности.

- Списъкът се сортира според честотата на срещане, като най-често срещаните са в началото, а най-рядко срещаните – в края.
- Списъкът се разделя на две части (бинарно дърво).
- На първата част се дава стойност нула, а на втората единица. Кодовете на всички символи от първата част започват с 0, а тези от втората – с единица.
- Рекурсивно се прилага процедурата, докато във всяка група не остане само един символ.

### **22.2.2 Метод на Хъфман**

Метода на Шанон-Фано бързи бива изместен от по-ефективния метод на Хъфман. По много неща Хъфмановото кодиране прилича на метода на Шанон-Фано. И при него биват създадени кодове с различна дължина, символите с по-голяма вероятност получават по-малки кодове и декодирането става чрез двоично дърво. Построяването на двоичното декодиращо дърво обаче е различно – Хъфмановото дърво се строи от листата към корена:

1. Намират се два свободни възела с най-малки тежести.
2. Създава се родител с тежест равна на сумата на тежестите на двета подвъзела.
3. Създаваният възел се добавя към списъка с възли, а двета подвъзли се премахват.

4. На единия от подвъзлите се присвоява единица, а на другия – 0.
5. Повтаря се рекурсивно, докато не остане само един възел.

### **22.2.3 Метод за подобreno Хъфманово кодиране, чрез адаптивния метод**

Малък недостатък на метода на Хъфман е нуждата от добавяне на таблица с вероятностите към компресирания файл. В противен случай декодиращата програма не би могла да състави правилен модел. Таблицата обикновено не е голяма по размер и не променя много крайния размер на компресирания файл, но с подобряването на компресията размерът ѝ значително започва да нараства. Адаптивното кодиране е решението на този проблем. Чрез него можем да използваме модели от по-висок ред без да нараства обема на статистиката. Принципа, на който се работи при адаптивното кодиране, е променяне на самото дърво въз основа на предходните данни и липса на данни за бъдещите статистики. Принципа на работа е следният:

- Кодиращата и декодиращата програма започват работа с идентични модели.
- След извеждането на първия символ от кодиращата програма се извършва update на модела.
- Програмата взима под внимание току що обработеният символ и променя таблицата с честотите съобразно.
- Съобразно се построява и ново Хъфманово дърво.

Тъй като построяването на ново дърво на всяка итерация (всеки нов символ) би довело до изключително бавно компресиран, се използва едно от свойствата на двоичните дървета – свойството на двойника. Всеки възел освен корена си има двойник – това е възела, който произлиза от същия родител. Едно дърво има свойство на двойника, ако възлите му могат да се изпишат по възходящ ред на теглата им и всеки възел е съседен с двойника си. Благодарение на това свойство обновяването на дървото става лесно – чрез поредица от промяна на теглата на възлите съобразно обработените символи и преместване на възлите, които не удовлетворяват въпросното свойство.

Още един начин за подобряване на компресията е да не се заделя място в таблиците за символи, които не участват във файла. При кодирането на Хъфман това е лесно, тъй като така или иначе работим с пълните таблици, но при адаптивното кодиране не знаем кой символ ще присъства и кой не. Това е и един от малкото минуси на адаптивното кодиране. Решение на този проблем съществува, но то не е приложимо във всички случаи.

Цялостен проблем на Хъфмановото кодиране е това, че съществуват оптимален брой битове за закодиране на даден символ, който често не е цяло число. При хъфмановото кодиране, обаче, е невъзможно да се съпоставят битове, които не са цяло число на даден символ. Например, ако чрез някакъв статистически метод сме изчислили, че вероятността на даден символ е 90%, то оптималната дължина на кода му е 0.15 бита. При Хъфмановото кодиране, този символ ще получи 1 бит, което е над 6 пъти повече, от оптималното. Заради този проблем, Хъфмановото кодиране е далеч от универсално.

#### **22.2.4 Аритметично кодиране**

Аритметичното кодиране решава гореописания проблем по много елегантен начин. При този тип кодиране се изоставя идеята на всеки символ да се съпоставя код. При аритметичното кодиране целият входен поток се заменя от едно единствено число (между 0 и 1) с плаваща запетая. Числото дава единствен резултат при декодирането му и това е входния поток, преминал през кодирането.

За генериране на числото отново се използва статистически метод, който съпоставя на даден символ или поредица от символи дадена вероятност.

На символите се съпоставят интервали от вероятностната крива, съобразно с изчислените на предната стъпка вероятности. Дължината на интервала за всеки символ е равна на вероятността той да бъде срепнат.

Най-важната част от входният поток е първият му символ. За да бъде правилно кодирането, числото, съдържащо кодираната информация, трябва да бъде в интервала на въпросния първи символ.

По нататък при кодирането интервала само се стеснява, като се добавят вероятностите на следващите символи, смалявайки интервала. Принципа на добавяне към числото е добавяне на вероятностите на последвалите символи към края му. Най-общо казано – нещо силно наподобяващо конкатенация на цифрите след десетичната запетая.

При декодиране се извършват обратните стъпки, само че интервала се уголемява с извлечането на всеки следващ символ. Поради ограниченността на някои видове хадуер по отношение на съхранението и обработката на числа с плаваща запетая е възможно трансформиране на алгоритъма за аритметично кодиране, така че вместо работи с число между 0 и 1, да се работи с цяло число в друг даден интервал.

#### **22.2.5 Кодиране чрез речници**

По-горните модели използват статистически методи за кодиране и компресирането става благодарение на намалената дължина на кодовете на най-често срепнатите символи. Заради това степента на компресия зависи от качеството на статистическия модел. При речниковите кодирания се използва различен подход – те не копират символи чрез кодове, а представят символните низове с променлива дължина с единични кодове. Кодовете, в случая, са индекси към речник (асоциативен масив) с низове. Ако дължината на използванния код е по-малка от дължината на заместения низ, получаваме компресия. Идеята на асоциативните масиви е пределно ясна и крайно елементарна. Съществуват два вида кодиране с речници – статично и адаптивно.

Статичен метод за кодиране с речници – когато работим с низове, които са краен брой, например елементи на база данни, съдържаща всички алкохолни напитки в заведение, е удачно да се създаде речник преди да се пристъпи към компресирането. Създаден един път, този речник би се използвал еднакво ефективно за кодирането и декодирането на информациите, като степента на компресия ще бъде доста голяма. Тези речници се наричат статични, поради това, че не се променят в процеса на компресиране. Предимство на този тип речници е това, че те са пригодени към типа данни, които ще бъдат компресирани. При по-малки файлове, необходимостта за предаване на речника към декомпресиращата програма

може да намали значително степента на компресия, което е основен недостатък на статичните речници.

Адаптивен метод за кодиране с речници – поради очевидните недостатъци на статичните речници (напр. трудно бихме кодирали Българския тълковен речник, чрез статични речници), по-широко употребявани са адаптивните речници. При тях компресирането се започва с малък или празен речник, като в процеса се добавят нови елементи, които по-късно ще се заместват с кодове.

#### **22.2.6 Адаптивни речници**

Адаптивните речници работят на следния принцип:

Входът се прочита на части, като се търсят съвпадения с фразите в речника. По време на сравнение може да се следи и за частични съвпадения.

Новите фрази / низове се добавят към речника.

Текста се кодира така, че те да бъдат лесно различими.

Декомпресирането се случва по обратния начин. В момента адаптивните речници са един от най-популярните методи за компресиране без загуба на информацията. Поради тази причина са се изграждали много подобрения върху алгоритмите за кодиране чрез този метод, които са увеличавали производителността и степента на компресия на информацията. Следните са едни от най-известните:

LZ77 – използва прочетения преди това текст като речник. Замества низовете с указатели към речника и по този начин постига компресирането на данните. Основна структура на LZ77 е прозорецът, който се състои от голям блок от насокро закодиран текст и буфер, който съдържа частта от текста непосредствено след текста, който бива кодиран в момента. Дължината на текстовия прозорец обикновено е няколко хиляди символа, а буфера – рядко над 100. Очевидни са предимствата на работенето с така наречените *sliding window*, но въпреки това LZ77 има доста сериозна пречка пред постигането на добра производителност. При кодирането трябва да се направи сравнение с буфера за всяка една позиция от прозореца. При декодиране този проблем не съществува, тъй като тогава само се копират фрази. Също така, за постигане на добра скорост е важно „плъзгането“ да се реализира по добър начин (добра индексация, указатели и пр.), а не просто постепенно обхождане на всеки символ.

LZSS – този метод е различен от LZ77 с това, че той съхранява текста в прозорците по структуриран и организиран начин, а не просто като един непрекъснат низ. Това става чрез двоично дърво, което намалява времето за търсете на дадена фраза, което от своя страна позволява да се експериментира с по-големи прозорци. Другата разлика е, че при LZSS е възможно смесването на указатели и обикновен текст, което евентуално може да доведе до намаляване на „служебната“ информация.

Съществуват и други методи, като LZ78, LZW, все алгоритми написани от Abraham Lempel и Jacob Ziv (от където и LZ префиксът), като в измислянето на последния има участие и Terry Welch, както и LZMA, LZWL, LZO, LZX, LZRW, LZJB, LZS, всички от които са част от

Lempel-Ziv семейството. Всеки от тях съдържа някаква промяна на оригиналния метод, на които, обаче, няма да се спра в тази курсова работа.

Съществуват и други сравнително популярни алгоритми за кодиране, които не са част от LZ семейството като DEFLATE – комбинация между LZ77 и Хъфманово кодиране; Byte-pair encoding, който се основава на изграждането на граматика в математическия смисъл на думата; Run-length encoding, който записва броя повторения на даден символ или поредица от символи и други.

### **22.2.7 Други методи за кодиране**

Съществуват и много други методи за кодиране, които няма да разгледам в детайли тук. Все пак за пълнота ще спомена голяма част от тях – Шанон-Фано-Елиас, който е предшественик на аритметичния метод; Кодиране на Голомб, Елиас-гама кодиране, Експоненциално Голомб кодиране, Левенщайн кодиране, Фиbonacci кодиране – алгоритми за кодиране, които са насочени предимно към кодиране на числа и разчитат на дадени математически свойства и формули.

## **22.3 Методи за компресия със загуба**

Съществуват и множество методи за компресия със загуба. Както споменах по-горе, те се използват предимно за компресиране на видео и аудио. Поради тази причина, те не би трябвало да попадат в обхвата на темата, тъй като не са особено важни за изкуствения интелект, като изключим някои специфични негови направления. Поради тази причина ще се спра съвсем кратко на тях.

### **22.3.1 Компресия на аудио**

При компресиране на аудио изпускането на част от информационния поток не води до съществени изменения в качеството, но води до значително намаляване на размера им. Благодарение на технологиите за компресия на аудио не само става възможен малкия размер на аудио файлове със сравнително добро качество, но се осигуряват и международните телефонни разговори и телефонията през интернет. Алгоритмите за компресия на аудио обикновено се възползват от несъвършенствата на човешкия слух – неспособността на човешкото ухо да възприема тонове с голяма разлика в амплитудите като такива, неспособността му да възприема като отделни тонове с разлика в честотата по-малка от 2Hz (това води до най-голяма компресия) и малкия диапазон на чувствителност на човешкото ухо. Например при MP3, честотната ента от 20Hz до 16kHz се разделя на 32 поддиапазона с оптимални стойности за минимална честотна девиация, амплитуден коефициент и амплитудна атенюация, като тези стойности вариират за различните bitrates. Популярни аудио формати са MP2, MP3, VQF, WMA и др.

### **22.3.2 Компресия на видео**

При компресията на видео се разчита на няколко похвата. Един от тях е изграждането на ключови кадри. Този похват разчита на това, че в голям брой кадри дадени области от екрана не се изменят. Следователно се създава ключов кадър, който съдържа оригиналното

изображение, докато то не се промени в по-нататъшен кадър. Така е нужно да се описват промените, които настъпват само между два ключови кадъра. За да не бъде насечена картина при възпроизвеждане се ограничават до даден малък размер областите, за които се изчислява промяната на картина и, съответно, ключовите кадри. Друг популяррен похват е разделянето на видео потока на отделни обекти – например фон и подвижни обекти (както детските рисувани филмчета от преди време). Фона се предава като отделен поток, който е компресиран с алгоритъм за статични изображения, а подвижните части се предават като независим поток, компресиран чрез друг метод за видео компресия. Същото важи и за аудио каналите, вървящи с видео потоците.

## **22.4 Резюме**

Компресирането на данни е тема, по която се пишат цели книги и учебници. Поради този факт, тази курсова работа далеч не изчерпва всички аспекти и детайли (особено детайли) около намаляването на обема на данни. Тук са описани по-важните аспекти на компресирането, някои важни понятия, видовете компресия, някои от по-значимите методи за кодиране (а други са само споменати). Важно е да се отбележи, че компресирането е жизнено важно за голяма част от софтуера, който ползваме ежедневно (включително операционните системи) и употребата му в областта на изкуствения интелект е една малка капка в морето на компресията на данни. А що се отнася до изкуствения интелект – намаляването на обема на данни е важно по очевидни причини – при работата с изкуствен интелект често се налага съхраняването на големи количества данни и докато структурите за представяне на знания, те не се справят с проблема с ограниченият обем на запаметявящите устройства. Именно този проблем намира приложение на компресирането в изкуствения интелект. Поради тази, а и други, причини можем само да се надяваме, че с подобряването на методите в областта на компресията ще се развива и изкуственият интелект, както и обратното.

## **22.5 Ресурси**

David Solomon - Data Compression: The Complete Reference, Springer, 2004

Ida Mengyi Pu - Fundamental Data Compression Butterworth-Heinemann, 2005