**3.4. Стратегії неінформованого пошуку**

У цьому розділі розглядаються п’ять стратегій пошуку, які відомі під назвою **неінформованого пошуку**(також називають **сліпим пошуком**). Цей термін означає, що в даних стратегіях не використовується додаткова інформація про стани, крім тієї, що подана в описі задачі. Все, на що вони здатні, - створювати спадкоємців і відрізняти цільовий стан від нецільового. Стратегії, що дозволяють виявити, чи є один нецільовий стан «більш багатообіцяючим» у порівнянні з іншим, називаються стратегіями **інформованого пошуку** **або евристичного пошуку**; вони розглядаються у розділі 4. Всі стратегії пошуку відрізняються тим, у якому порядку відбувається розгортання вузлів.

**Пошук в ширину**

**Пошук в ширину –** це проста стратегія, в якій спочатку розгортається кореневий вузол, потім – всі спадкоємці кореневого вузла, потім спадкоємці спадкоємців і т.д. Тобто при пошуку в ширину, перед тим як розгорнути вузли на наступному рівні, розгортаються всі вузли на даній глибині дерева пошуку.

Пошук в ширину можу бути реалізовано шляхом виклику процедури Tree-Search з пустою периферією, яка представляє собою послідовну чергу(First-In-First-Out --- FIFO), що гарантує, що спочатку будуть розгорнуті вузли, що відвідуються першими. Інакше кажучи, до організації пошуку в глибину призводить виклик процедури Tree-Search(problem, FIFO-Queue()). В черзі FIFO передбачена вставка всіх знову сформованих спадкоємців в кінець черги, а це означає, що поверхневі вузли розгортаються раніше ніж глибокі. На рисунку 3.7 показано хід пошуку в простому бінарному дереві.

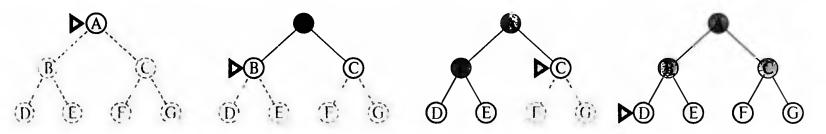


Рис. 3.7. Пошук в ширину в простому бінарному дереві. На кожному етапі вузол, що має бути розгорнуто наступним, позначено маркером

Проведемо оцінку пошуку в ширину з використанням чотирьох критеріїв, описаних в минулому розділі. Очевидно, що цей пошук є повним – якщо найбільш поверхневий вузол знаходиться на деякій кінцевій глибині *d*, то пошук в ширину в кінцевому рахунку дозволяє його знайти після розгортання більш поверхневих вузлів(за умови, що коефіцієнт розгалуження *b* являється кінцевим). Найбільш поверхневий цільовий вузол не обов’язково є оптимальним; формально пошук в ширину буде оптимальним, якщо ціна шляху виражається у вигляді неспадної функції глибини вузла. (Наприклад, таке припущення виправдається, якщо всі дії мають однакову ціну.)

До сих пір наданий вище опис пошуку в ширину не прогнозував ніяких неприємностей. Але така стратегія не завжди є оптимальною; щоб зрозуміти, з чим це пов’язано, треба визначити, яка кількість часу і який об’єм пам’яті потрібен для виконання пошуку. Для цього розглянемо гіпотетичний простір станів, у якому кожен стан має *b* спадкоємців. Корінь цього дерева породжує *b* вузлів на першому рівні, кожен з яких породжує ще b вузлів на другому рівні, що відповідає загальній кількості вузлів, що дорівнює *b2* . Кожен з них породжує ще b вузлів на третьому рівні, що відповідає загальній кількості вузлів, що дорівнює *b3* і т.д. А тепер припустімо, що рішення знаходиться на глибині d. В найгіршому випадку на рівні d необхідно розгорнути всі вузли, крім останньому (адже сам цільовий вузол не розгортається), що призводить до обробки *bd+1-b* вузлів на рівні *d+1*. Це означає, що загальна кількість оброблених вузлів дорівнює:

*b + b2 + b3 + … bd + (bd+1 -b) = O(bd+1)*

Кожен оброблений вузол має залишатися в пам’яті, адже він або відноситься до периферії, або є предком периферійного вузла. Так, просторова складність стає такою ж, як і часова (з урахування додавання одного вузла, що відповідає кореню).

Тому досліджувачі, що проводять аналіз складності алгоритму, засмучуються(або радіють, якщо їм подобається долати складнощі), стикнувшись з експоненціальними оцінками складності, такими як O(bd+1). В табл. 3.1 показано, з чим це пов’язано. В ній наведені вимоги до часу та до об’єму пам’яті при використанні пошуку в ширину з коефіцієнтом b=10 для різних значень глибини рішення d. При заповненні цієї таблиці припускалося, що в секунду може бути сформовано 10 000 вузлів, а для кожного вузла потрібно 1000 байтів пам’яті. Цим припущенням приблизно відповідають багато задач пошуку при їх рішенні на сучасному ПК (з урахуванням підвищувального або понижуючого коефіцієнта 100).

На основі табл. 3.1 можна зробити два важливих висновки. Перш за все, *при пошуку в ширину найбільш складною проблемою у порівнянні з значним часом виконання є забезпечення необхідності в пам’яті.* Затрати часу рівні 31 годині не здаються такими значними при очікуванні рішення важливої задачі з глибиною 8, але дуже мало комп’ютерів мають терабайт оперативної пам’яті, який потрібен для цього. На щастя, існують інші стратегії пошуку, які потребують менше пам’яті.

Другий висновок полягає в тому , що вимоги до часу все ще залишаються важливим фактором. Якщо задачу, що розглядається, має рішення на глибині 12, то (з урахуванням зроблених припущень) знадобиться 35 років на пошук в ширину(а насправді на будь-який неінформований пошук), щоб знайти її рішення. Узагалі, *задачі пошуку з експоненціальною складністю неможливо вирішити за допомогою неінформованих методів в усіх екземплярах задач, крім найменших.*

**Таблиця 3.1. Потреби в часі та об’ємі пам’яті для пошуку в ширину. Наведені тут дані отримані за наступних припущень: коефіцієнт розгалуження – b=10; швидкість формування вузлів – 10 000 вузлів/ секунда; об’єм пам’яті 1000 байтів/вузол.**

|  |  |  |  |
| --- | --- | --- | --- |
| Глибина | Кількість вузлів | Час | Пам’яті |
| 2 | 1100 | 0,11 секунди | 1 мегабайт |
| 4 | 111 100 | 11 секунд | 106 мегабайт |
| 6 | 107 | 19 хвилин | 10 гігабайтів |
| 8 | 109 | 31 година | 1 терабайт |
| 10 | 1011 | 129 днів | 101 терабайт |
| 12 | 1013 | 35 років | 10 петабайтів |
| 14 | 1015 | 3523 роки | 1 ексабайт |

**Пошук по критерію вартості**

Пошук в ширину є оптимальним, якщо вартості всіх етапів рівні, адже в ньому завжди розгортається найбільш поверхневий нерозгорнутий вузол. З допомогою простого доповнення можна створити алгоритм, який є оптимальним при будь-якій функції визначення вартості етапу. Замість розгортання найбільш поверхневого вузла пошук **по критерію вартості** забезпечує розгортання вузла n з найменшою вартістю шляху. Зверніть увагу на те, що якщо вартості всіх етапів рівні, такий пошук ідентичний пошуку в ширину.

При пошуку по критерію вартості враховується не кількість етапів, що належать шляху, а тільки їх сумарна вартість. Тому процедура цього пошуку може увійти в нескінчений цикл, якщо виявиться, що в ній розгорнуто вузол, який має дію з нульовою вартістю, яка знову вказує на той самий стан (наприклад, дія NoOp). Можна гарантувати повноту пошуку за умови , що вартість кожного етапу більше або рівна деякої невеликої додатної константи *e.* Ця умова є також достатньою для забезпечення оптимальності. Це означає, що вартість шляху завжди зростає з мірою проходження по цьому шляху. Тому перший цільовий вузол, вибраний для розгортання, являє собою оптимальне рішення. (Нагадаємо, що в процедурі Tree-Search перевірка цілі здійснюється лише до вузлів, вибраних для розгортання.) Рекомендуємо читачу спробувати використати алгоритм для пошуку найкоротшого шляху до Бухареста.

Пошук по критерію вартості направляється з урахуванням вартостей шляхів, а не значень глибини в дереві пошуку, тому його складність не може бути легко охарактеризованою в термінах *b* та *d*. Замість цього припустімо, що С\* - вартість оптимального рішення, і припустімо, що вартість кожної дії дорівнює щонайменше *e.* Це означає *,* що часова та просторова складність цього алгоритму в гіршому випадку дорівнює O(b1+⌊C\*/ɛ ⌋), тобто може бути набагато більшою ніж bd. Це пов’язано з тим, що процедури пошуку по критерію вартості можуть і часто виконують перевірку великих дерев, що складаються з малих етапів, перед тим як перейти до дослідження шляхів, у які входять крупні, але, можливо, більше корисні етапи. Безумовно, якщо всі вартості етапів рівні, то b1+⌊C\*/ɛ ⌋ дорівнює bd.

**Пошук в глибину**

При **пошуку в глибину** завжди розгортається найбільш глибокий вузол в поточній периферії дерева пошуку. Хід такого пошуку показано на рис. 3.8. Пошук безпосередньо переходить на найбільш глибокий рівень дерева пошуку, на якому вузли не мають спадкоємців. По мірі того як ці вузли розгортаються вони видаляються з периферії, тому в подальшому пошук «оновлюється» з наступного найбільш поверхневого вузла, який все ще має недосліджених спадкоємців.

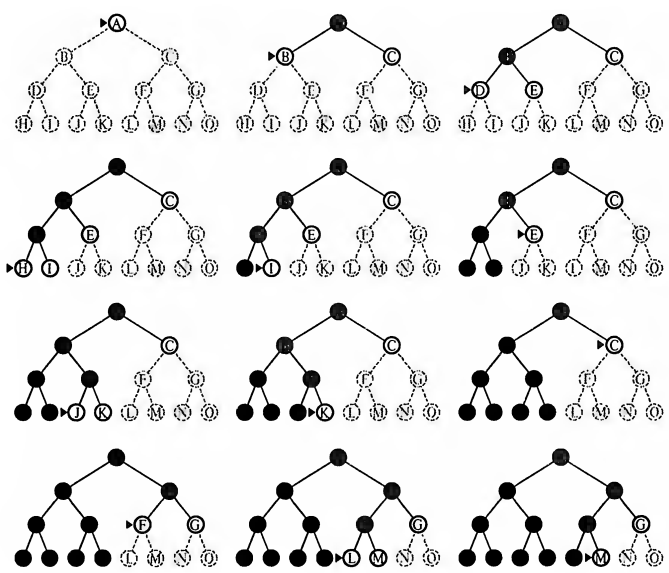


Рис. 3.8. Пошук в глибину в бінарному дереві. Вузли, які було розгорнуто і не маються спадкоємців в цій периферії, можуть бути видалені з пам’яті; ці вузли позначені чорним кольором. Припускається, що вузли на глибині 3 не мають спадкоємців і єдиним цільовим вузлом є М.

Ця стратегія може бути реалізована в процедурі Tree-Search за допомогою черзі LIFO(Last-In-First-Out), яку ще називають *стеком*. В якості альтернативи способу реалізації на основі процедури Tree-Search пошук в глибину часто реалізують за допомогою рекурсивної функції, що викликає сама себе в кожному з дочірніх вузлів по черзі. (Рекурсивний алгоритм пошуку в глибину, в якому передбачено границю глибину, наведено в лістингу 3.4.)

Пошук в глибину має дуже скромні вимоги до пам’яті. Він вимагає зберігання тільки єдиного шляху від кореня до листка, поряд з нерозгорнутими сестринськими вузлами, що залишилися, для кожного вузла шляху. Після того як було розгорнуто деякий вузол, він може бути видаленим з пам’яті, адже скоро будуть повністю досліджені всі його спадкоємці (див. рис. 3.8). Для простору станів з коефіцієнтом розгалуження b і максимальною глибиною m пошук в глибину вимагає зберігання лише *bm+1* вузлів. Використовуючи такі ж припущення, як і в табл. 3.1, і припускаючи, що вузли, що знаходяться на тій же глибині, що і цільовий вузол, не мають спадкоємців, автори визначили, що на глибині d=12 для пошуку в глибину потрібно лише 118 кілобайтів замість 10 петабайтів, тобто необхідність в об’ємі зменшується приблизно в 10 мільярдів раз.

В одному з варіантів пошуку в глибину, що називають **пошуком з поверненнями**, використовується ще менше пам’яті. При пошуку з поверненнями кожен раз формується лише один спадкоємець, а не всі спадкоємці; у кожному частково розгорнутому вузлі запам’ятовується інформація про те, який спадкоємець має бути сформовано наступним. Таким чином, треба лише О(m) пам’яті, а не О(bm). При пошуку з поверненнями використовується ще один прийом, що дозволяє економити пам’ять(і час); ідея у тому, щоб при формування спадкоємця має безпосередньо модифікуватися опис теперішнього стану, а не викликатися його попереднє копіювання. При цьому необхідність у пам’яті зменшується до об’єму, який потрібен для зберігання тільки одного опису стану і О(m) дій. Але для успішного використання цього прийому треба мати можливість відміняти кожну модифікацію при поверненні, виконаному для формування наступного спадкоємця. При вирішення задач з об’ємними описами станів, таких як роботизована збірка, використання вказаних методів модифікації станів стають найважливішим фактором успіху.

Недоліком пошуку в глибину є те, що в ньому може бути зроблено неправильний вибір і перехід в тупикову ситуацію, пов’язану з проходженням вниз по дуже довгому(або навіть нескінченному) шляху, причому другий варіант міг би привести до рішення, що знаходиться далеко від кореня дерева. Наприклад, на рис. 3.8 пошук в глибину вимагав би дослідження всього лівого піддерева, навіть якби цільовим вузлом був вузол С, що знаходиться в правому піддереві. А якби цільовим вузлом також був вузол J, менш прийнятний у порівнянні з вузлом С, то пошук в глибину повернув би в якості рішення саме його; це означає, що пошук в глибину не є оптимальним. Крім того, якби ліве піддерево мало необмежену глибину, але не вміщало в собі рішень, то пошук в глибину так ніколи б і не закінчився; це означає, що даний алгоритм – неповний. В найгіршому випадку пошук в глибину формує всі О(bm) вузлів у дереві пошуку, де m – максимальна глибина будь-якого вузла. Варто зазначити, що m може бути набагато більшим у порівнянні з d (глибиною найбільш поверхневого рішення) і є нескінченним, якщо дерево має нескінченну глибину.

**Пошук з обмеженнями в глибині**

Проблему необмежених дерев можна вирішити, передбачивши використання під час пошуку в глибину заздалегідь визначеної границі глибини £. Це означає, що вузли на глибині розглядаються таким чином, ніби якби вони не мають спадкоємців. Такий підхід називається **пошуком з обмеженням в глибині.** Використання обмеження глибину дозволяє вирішити проблему нескінченого шляху. На жаль, у цього підході також уводиться додаткове джерело неповноти, якщо буде обрано значення £<d, інакше кажучи, якщо найбільш поверхнева ціль виходить за границі глибини. (Така ситуація є цілком ймовірною, якщо значення d невідомо.) Крім того, пошук з обмеженнями глибину буде неоптимальним при виборі значення £>d. Його часова складність дорівнює О(b£), а просторова складність – О(b\*£). Пошук в глибину може бути розглянуто як окремий випадок пошуку з обмеженням глибину при якому £=∞∞.

Іноді вибір границь глибини може бути засновано на кращому розумінні задачі. Наприклад, припустимо, що на карті Румунії, що розглядається, маємо 20 міст. Тому відомо, що якщо рішення існує, то воно повинно мати довжину не більше 19; це означає, що одним з можливих варіантів є £=19. Але в дійсності при уважному вивченні карти можна помітити, що будь-яке місто може бути досягнуто з будь-якого іншого міста не більше ніж за 9 етапів. Це число, відоме як діаметр простору станів, дає нам кращу границю глибини, яка веде до найбільш ефективного пошуку з обмеженням глибини. Але в більшості задач прийнятна границя глибини залишається невідомою, поки не буде вирішена сама задача.

Пошук з обмеженням глибини може бути реалізовано як проста модифікація загального алгоритму пошуку в дереві, або рекурсивного алгоритму пошуку в глибину. Псевдокод реалізації рекурсивного пошуку з обмеженням глибини наведено в лістингу 3.4. Зверніть увагу на те, що пошук з обмеженнями глибини може призвести до невдалих завершень двох типів: стандартне значення failure вказує на відсутність рішення, а значення cutoff говорить про те, що на даній границі рішення нема.

**Лістинг 3.4. Рекурсивна реалізація пошуку з обмеженням глибини**

***Function Depth-Limited-Search (problem, limit) returns рішення result або індикатор невдачі failure\cutoff***

***Return Recursive-DLS(Make-Node(Initial-State[problem]),***

***Problem, limit)***

***Function Recursive-DLS(node, problem, limit) returns рішення result або індикатор невдачі failure\cutoff***

***cutoff\_occurred?*** *←* ***неправдиве значення***

***if Goal-Test[problem](State[node]) then return Solution(node)***

***else if Deptbh[node] = limit then return індикатор невдачі cutoff***

***else for each спадкоємець successor in Expand(node, problem) do***

***result*** *← Recursive-DLS(successor, problem, limit)*

***if result = cutoff then cutoff\_occured?*** *← правдиве значення*

***else if result != failure then return рішення result***

***if cutoff\_occurred?***

***Then return індикатор невдачі cutoff***

***Else return індикатор невдачі failure***

**Пошук в глибину з ітеративним заглибленням**

**Пошук з ітеративним заглибленням** (або, точніше, пошук в глибину з ітеративним заглибленням) являє собою загальну стратегію, що часто використовується разом з пошуком в глибину, яка дозволяє знайти кращу границю глибини. Це досягається шляхом поступового збільшення границі(яка на початку стає , потім 1 потім 2 і т.д.) поки не буде знайдена нова ціль. Така подія відбувається після того, як границя глибини досягає значення d, глибини найбільш поверхневого цільового вузла. Цей алгоритм наведено в лістингу 3.5. В пошуку з ітеративним заглибленням поєднуються переваги пошуку в глибину і пошуку в ширину. Як і пошук в глибину, він характеризується малими вимогами у пам’яті, а саме, значенням О(bd). Як і пошук в ширину, він є повним, якщо коефіцієнт розгалуження є кінцевим оптимальним, якщо вартості всіх шляхів являються собою неспадну функцію глибини вузла. На рис. 3.9 показано чотири ітерації використання процедури Iterative-Deepening-Search до бінарного дерева пошуку, де рішення знайдено на четвертій ітерації.

**Лістинг 3.5. Алгоритм пошуку з ітеративним заглибленням, в якому повторно виконується пошук з обмеженням глибини при послідовному збільшенні границі. Він завершує свою роботу після того, як знаходиться рішення, або процедура пошуку з обмеженням глибини повертає значення failure, а це означає, що рішення не існує.**

***function*** *Iterative-Deepening-Search(problem)**returns рішення result або індикатор невдачі failure*

***inputs:*** *problem, задачі* ***for*** *depth ← 0* ***to*** *∞* ***do***

*result ← Depth-Limited-Search(problem, depth)*

***if*** *result != cutoff* ***then return*** *рішення result*

Пошук з ітеративним заглибленням може на перший погляд знатися марнотратним, адже одні й ті самі формування формуються декілька разів. Але як виявилось, такі повторні операції є не занадто затратними. Причина цього у тому, що в дереві пошуку з одним і тим самим(або майже тим самим) коефіцієнтом розгалуження на кожному рівні більшість вузлів знаходяться на нижньому рівні, тому не має великого значення те, що вузли на верхніх рівнях формуються багатократно. В пошуку з ітеративним заглибленням вузли на нижньому рівні (з глибиною d) формуються один раз, ті вузли, які знаходяться на попередньому формуються двічі, і т.д., поки дочірніх вузлів кореневого вузла, які формуються d разів. Тому загальна кількість вузлів, що формується виражається формулою:

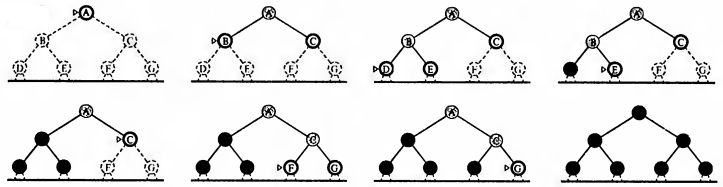
N(IDS) = (d)b + (d-1)b2 + … + (1)bd

яка відповідає часовій складності порядка О(bd). Цю кількість можна порівняти з кількістю вузлів, що формується при пошуку в ширину:

N(BFS) = b + b2 + … + bd + (bd+1-b)

Границя 0: 

Границя 1: 

Границя 2: 

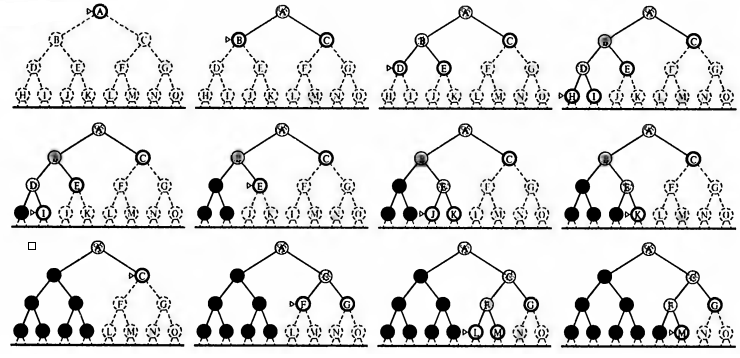
Границя 3: 

Рис. 3.9. Чотири ітерації пошуку з ітеративним заглибленням в бінарному дереві

Варто підмітити, що при пошуку в ширину деякі вузли формуються на глибині d+1, а при ітеративному заглибленні цього не відбувається. Результатом є те, що пошук з ітеративним заглибленням фактично виконується швидше, ніж пошук в ширину, незважаючи на повторне формування станів. Наприклад, якщо b=10, d=5, то відповідні оцінки кількості вузлів приймають наступні значення:

N(IDS) = 50+400+3000+20 000+100 000=123 450

N(BFS) = 10+100+1000+10 000+100 000+999 990 = 1 111 100

*Взагалі, ітеративне заглиблення – це кращий метод неінформованого пошуку за тих умов, коли змінюється великий простір пошуку, а глибина рішення невідома.*

Пошук з ітеративним заглибленням аналогічний пошуку в ширину в тому відношенні, що в ньому при кожній ітерації перед переходом на наступний рівень досліджується повний рівень нових вузлів. На перший погляд може здатися доцільною розробка ітеративного аналога пошуку по критерію вартості, який би успадкував би від останнього алгоритму гарантію оптимальності, дозволяючи разом з ти виключити його високі вимоги до пам’яті. Ідея у тому, щоб замість збільшення границь глибини використовувались зростаючі границі вартості шляху. Результуючий алгоритм отримав назву пошук з ітеративним подовженням, розглядається у вправі 3.11. Але, на жаль, було встановлено, що пошук з ітеративним подовженням характеризується більш важливими витратами, ніж пошук по критерію вартості.

**Двонаправлений пошук**

В основі двонаправленого пошуку лежить така ідея, що можна одночасно проводити два пошуки (в прямому направлені, від початкового стану, і в оберненому, від цілі), зупиняючись після того, як два процеси пошуку зустрінуться на середині (рис. 3.10). Справа в тому, що значення bd/2+bd/2 набагато менше, ніж bd, або, як показано на малюнку, площа двох невеликих кругів менше площі одного великого круга з центром в початку пошуку, який охоплює ціль пошуку.

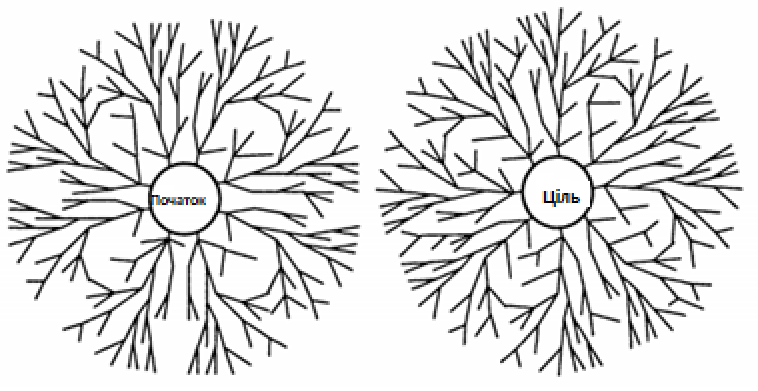


Рис. 3.10. Схематичне зображення двонаправленого пошуку в тому стані, коли він має скоро завершитися успішно після того як одна з гілок, що походить з початкового вузла, зустрінеться з гілкою з цільового вузла.

Двонаправлений пошук реалізовується за допомогою метода, в якому передбачено перевірка в одному чи в обох процесах пошуку кожного вузла перед його розгортанням для визначення того, чи не знаходиться він на периферії іншого дерева пошуку; у випадку позитивного результату перевірки - рішення знайдено. Наприклад якщо задача має рішення на глибині d=6 і в кожному направленні виконується пошук в ширину з послідовним розгортанням по одному вузлу, то в найгіршому випадку ці два процеси зустрінуться, якщо в кожному з них буде розгорнуто всі вузли на глибині 3, крім одного. Це означає, що при b=10 буде сформована загальна кількість вузлів, що дорівнює 22 000, а не 11 111 100, як при стандартному пошуку в ширину. Перевірка належності вузла до іншого дерева пошуку може бути виконана за постійний час за допомогою хеш-таблиці, тому часова складність двонаправленого пошуку визначається як О(bd/2) . В пам’яті необхідно зберігати хоча б одне з дерев пошуку, щоб можна було виконати перевірку належності до іншого дерева, тому просторова складність також визначається як О(bd/2). Такі вимоги до простору є одним з найбільш важливих недоліків двонаправленого пошуку. Цей алгоритм є повним і оптимальним (при однакових вартостях етапів) , якщо обидва процеси пошуку виконуються в ширину; інші поєднання методів можуть характеризуватися відсутністю повноти, оптимальності або того й іншого одночасно.

Завдяки такому зменшення часової складності двонаправлений пошук стає досить привабливим, але як організувати пошук в зворотному пошуку? Це не так легко, як здається на перший погляд. Припустімо, що батьками вузла n, що визначаються функцією Pred(n), є всі ті вузли, для яких n є спадкоємцем. Для двонаправленого пошуку потрібно, щоб функція визначення батьків Pred(n) була ефективною. Найпростішим є той випадок, коли всі дії в просторі станів можуть бути обернені таким чином, що Pred(n)=Succ-1(n), а інші випадки вимагають проявити значну винахідливість.

Розглянемо питання з тим, що мається на увазі під поняттям «ціль» при пошуку «в зворотному направленні від цілі». В задачах гри в вісім і пошуку маршруту в Румунії є лише один цільовий стан, тому зворотній пошук нагадує прямий пошук. Якщо ж є декілька явно перерахованих цільових станів(наприклад, два показаних на рис. 3.2 цільових стани, в яких квадрати поля не вміщають сміття), то може бути створено новий фіктивний цільовий стан. Інакше кажучи, формування деяких надлишкових вузлів можна уникнути, розглядаючи множину цільових станів як єдиний цільовий стан, кожен з батьків якого також є множиною станів, а саме, множина станів, що має відповідного спадкоємця в множині цільових станів (див. розділ 3.6).

Найбільш складним випадком для двонаправленого пошуку є така задача, в якій для перевірки цілі дано лише неявний опис деякої (можливо більшої) множини цільових станів, наприклад всіх станів, що відповідає перевірці цілі «мат» в шахах. При зворотному пошуку знадобилось би створити компактні описи всіх станів, які дозволяють поставити мат за допомогою ходу m1 тощо; і всі ці описи треба було б звіряти з станами, що формуються при прямому пошуку. Загального ефективного способу рішення такої проблеми не існує.

**Порівняння стратегій неінформованого пошуку** В табл. 3.2 наведено порівняння стратегій пошуку в термінах чотирьох критерії оцінки, сформованих в розділі 3.4.

**3.5. ЗАПОБІГАННЯ ФОРМУВАННЯ СТАНІВ, ЩО ПОВТОРЮЮТСЯ**

Аж до цього моменту ми розглядали всі аспекти пошуку, але ігнорували одне з найбільш важливих ускладнень в процесі пошуку — ймовірність появи непродуктивних витрат часу при розгортанні станів, які вже зустрічалися і були розгорнуті перед цим. При вирішенні деяких завдань така ситуація ніколи не виникає; в них простір станів являє собою дерево і тому є тільки один шлях до кожного стану. Зокрема, ефективним є формулювання завдання з вісьмома ферзями (в якому кожен новий ферзь поміщається на лівий порожній вертикальний ряд), його ефективність в значній мірі обумовлена саме цим – кожен стан може бути досягнуто тільки по одному шляху. А якби завдання з вісьмома ферзями була сформульована таким чином, що будь-якого ферзя дозволялося б ставити на будь-яку вертикаль, то кожного стану з *n* ферзями можна було б досягти за допомогою *n*! різних шляхів.

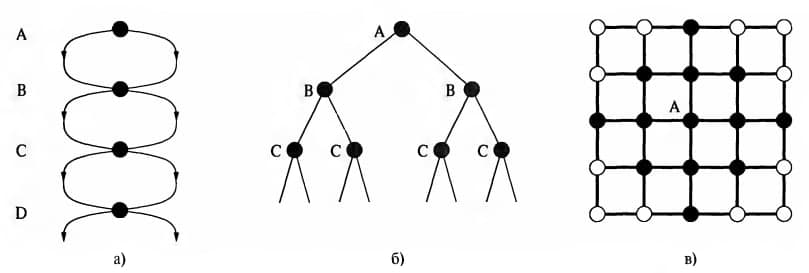
**Таблиця 3.2. Оцінка стратегій пошуку, де b — коефіцієнт розгалуження; d — глибина найбільш поверхневого рішення; m — максимальна глибина дерева пошуку; l — межа глибини. Застереження, позначені малими буквами, означають наступне: а-повний, якщо коефіцієнт розгалуження b кінцевий; б — повний, якщо вартість кожного етапу ≥*Ɛ* при деякому додатному значенні *Е*; в — оптимальний, якщо вартості всіх етапів є однаковими; г — можна застосовувати, якщо в обох напрямках здійснюється пошук в ширину.**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Характеристика** | **Пошук в ширину** | **Пошук за критерієм вартості** | **Пошук в глибину** | **Пошук з обмеженням глибини** | **Пошук з ітерактивним поглибленням** | **Двохнаправлений пошук (якщо він застосуємий )** |
| **Повнота** | **Така** | **Така,б** | **Ні** | **Ні** | **Така** | **Така,г** |
| **Тимчасова складність** | ***О(bd+1)*** | ***О(b1+[c\*/Ɛ])***  **(*[]*округлення вниз)** | ***О (bm)*** | ***О (bl)*** | ***О (bd)*** | ***О (bd/2)*** |
| **Простороваскладність** | ***О(bd+1)*** | ***О(b1+[c\*/Ɛ])***  **(*[]*округлення вниз)** | ***О(bm)*** | ***О (bl)*** | ***О (bd)*** | ***О (bd/2)*** |
| **Оптимальність** | **Такв** | **Так** | **Ні** | **Ні** | **Такв** | **Такв,г** |

У деяких задачах повторення станів є неминучим. До таких відносяться всі задачі, в яких дії є зворотними, такі як завдання пошуку маршруту і гри з легкими фішками. Дерева пошуку для цих проблем нескінченні, але якщо ми відітнемо деякі з повторюваних станів, то зможемо зменшити дерево пошуку до кінцевого розміру, формуючи тільки ту частину дерева, яка охоплює весь граф простору станів. Розглядаючи тільки частину дерева пошуку аж до деякої постійної глибини, можна легко виявити ситуації, в яких видалення повторюваних станів дозволяє досягти експоненціального зменшення вартості пошуку. В крайньому випадку простір станів розміром d + 1 (Рис. 3.11, а) стає деревом з 2d листям (Рис. 3.11, б). Більш реалістичним прикладом може служити Δ прямокутні грати, як показано на рис. 3.11, в. В решітці кожен стан має чотирьох наступників, тому дерево пошуку, що включає повторювані стани, має 4d листків, але існує приблизно тільки 2d2 різних станів з d етапами досягнення будь-якого конкретного стану. Для d = 20 це означає, що існує близько трільйона вузлів, але лише приблизно 800 різних станів.

Таким чином, нездатність алгоритму виявляти повторювані стани може послужити причиною того, що розв'язувана задача стане нерозв'язуваною. Таке виявлення зазвичай зводиться до того, що вузол, що підлягає розгортанню, порівнюється з тими вузлами, які вже були розгорнуті; якщо виявлено збіг, то алгоритм розпізнав наявність двох шляхів в один і той же стан і може відкинути один з них.

При пошуку в глибину в пам'яті зберігаються тільки ті вузли, які лежать на шляху від кореня до поточного вузла. Порівняння цих вузлів з поточним вузлом дозволяє алгоритму виявити циклічні шляху, які можуть бути негайно відкинуті. Це дозволяє забезпечити, щоб кінцеві простори станів не перетворювалися в нескінченні дерева пошуку через цикли, але, на жаль, не дає можливості запобігти експоненціальному розростанню нециклічних шляхів в задачах, подібних наведеним на рис. 3.11. Єдиний спосіб запобігання цьому полягає в тому, що в пам'яті потрібно зберігати більше вузлів. У цьому полягає фундаментальний компроміс між простором і часом. *Алгоритми, які забувають свою історію, приречені на те, щоб її повторювати.*



*Рис. 3.11. Простори станів, які формують експоненційно більш крупні дерева пошуку: простір станів, в якому є дві можливих дії, що ведуть від А до В, два - від В до С і т.д .; це простір станів містить d +1 станів, де d - максимальна глибина (а); дерево пошуку, яке має 2d гілок, відповідних 2d шляхам через цей простір (б); Простір станів у вигляді прямокутної решітки (в); стану, що знаходяться в межах 2 етапів від початкового стану (А), позначені сірим кольором*

Якщо деякий алгоритм запам'ятовує кожен стан, який він відвідав, то може розглядатися як безпосередньо граф, що досліджує, простору станів. Зокрема, можна модифікувати загальний алгоритм *Tree-Search*, щоб включити в нього структуру даних, звану **закритим списком**, в якому зберігається кожен розгорнутий вузол. (Периферію, що складається з нерозгорнутих вузлів, іноді називають **відкритим списком**.) Якщо поточний вузол збігається з будь-яким вузлом із закритого списку, то не розгортається, а відкидається. Цьому новому алгоритму присвоєно назву алгоритму пошуку в графі, *Graph-Search* (лістинг 3.6). При вирішенні завдань з багатьма періодичними станами алгоритм *Graph-Search* є набагато більш ефективним в порівнянні з *Tree-Search*. У найгіршому випадку запропоновані їм вимоги до часу і простору пропорційні розміру простору станів. Ця величина може виявитися набагато меншою, ніж О (bd).

Питання в тому, чи оптимальний пошук в графі, залишається складним. Вище було зазначено, що поява повторюваного стану відповідає виявленню алгоритмом двох шляхів в один і той же стан. Алгоритм *Graph-Search*, наведений у лістингу 3.6, завжди відкидає знову виявлений шлях і залишає початковий; очевидно, що якщо цей знову виявлений шлях коротше, ніж первинний, то алгоритм *Graph-Search* може втратити оптимальне рішення. На щастя, можна показати (впр. 3.12), що цього не може статися, якщо використовується або пошук за критерієм вартості, або пошук в ширину з постійними цінами етапів; таким чином, ці дві оптимальні стратегії пошуку в дереві є також оптимальними стратегіями пошуку в графі. При пошуку з ітеративним поглибленням, з іншого боку, використовується розгортання в глибину, тому цей алгоритм цілком може пройти до деякого вузла по неоптимальному шляху, перш ніж знайти оптимальний. Це означає, що при пошуку в графі з ітеративним заглибленням необхідно перевіряти, чи не є знову виявлений шлях до вузла кращим, ніж первинний, і в разі позитивної відповіді в ньому може знадобитися переглядати значення глибини і вартості шляхів для нащадків цього вузла.

**Лістинг 3.6. Загальний алгоритм пошуку в графі. Множина сlosed може бути реалізована за допомогою хеш-таблиці для забезпечення ефективної перевірки повторюваних станів. У цьому алгоритмі передбачається, що перший знайдений шлях до стану s є найменш дорогим (див. текст)**

***Function*** *Graph-Search (problem, fringe)* ***returns*** *рішення*

*або індекатор невдачі failure*

*closed ← порожня множена*

*fringe ← Insert (Make – Node (Initial – State [problem]), fringe)*

***loop do***

***If*** *Empty? (fringe)* ***then return*** *індекатор невдачі failure*

*node ← Remove – First (fringe)*

***If*** *Goal – Test [problem] (State [node]) then return*

*Solution (node)*

***If*** *State [node] не перебуває у безлічі closed* ***then***

*додати State [node] до безлічі closed*

*Fringe ← Insert – All (Expand (node, problem), fringe)*

Між іншим, використання закритого списку сlоsed означає, що пошук в глибину і пошук з ітеративним поглибленням більше не мають лінійних вимог до простору. Оскільки в алгоритмі Graph-Search кожен вузол зберігається в пам'яті, деякі методи пошуку стають неможливими через недостатній обсяг пам'яті.