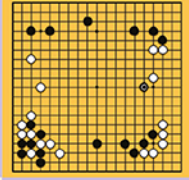
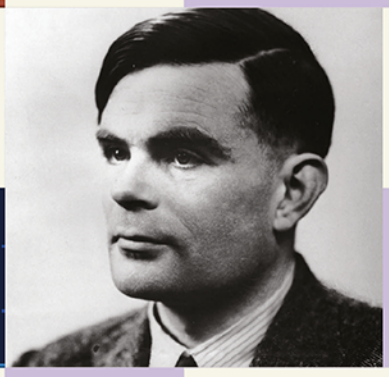
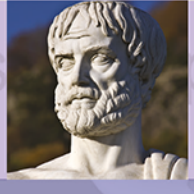






Stuart **Russell**

Peter **Norvig**

Sztuczna inteligencja

Nowe spojrzenie

Wydanie IV. Tom 1

Helion

Tytuł oryginału: Artificial Intelligence: A Modern Approach, 4th Edition

Tłumaczenie: Andrzej Grażyński

ISBN: 978-83-283-7608-3

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/szti41
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

SPIS TREŚCI

IV Wnioskowanie w warunkach niepewności

ROZDZIAŁ 3

ROZWIĄZYWANIE PROBLEMÓW ZA POMOCĄ WYSZUKIWANIA

W tym rozdziale wyjaśniamy, jak agent może poszukiwać ciągu akcji, które mają doprowadzić go do założonego celu.

Gdy prawidłowa akcja, którą powinien podjąć agent, nie jest dla niego od razu oczywista, zmuszony zostaje on do *planowania z wyprzedzeniem*, czyli rozważenia ciągu akcji formujących ścieżkę prowadzącą do stanu, który oznacza spełnienie celu. Agenta działającego w taki sposób nazywamy **agentem rozwiązującym problemy** (ang. *problem-solving agent*), a kryjące się za nim procesy obliczeniowe określamy mianem **wyszukiwania** (ang. *search*).

Opisywany w tym rozdziale agent rozwiązujący problemy zrealizowany jest w reprezentacji **atomowej** (patrz sekcja 2.4.7) — stany postrzeganego przezeń środowiska są niepodzielne, nie objawiają jakiejkolwiek struktury dla algorytmów rozwiązywania problemów. Agentów wykorzystujących reprezentację **czynnikową** lub **strukturalną** — zwanych **agentami planowania** — opisujemy w rozdziałach 7. i 11.

Omówimy kilka algorytmów wyszukiwania, ograniczając się do najprostszego typu środowiska — epizodycznego, jednoagentowego, w pełni obserwowalnego, deterministycznego, statycznego, dyskretnego i rozpoznanego. Odróżniać będziemy algorytmy **poinformowane** (ang. *informed*) od **niedoinformowanych** (ang. *uninformed*) — agent używający tych pierwszych ma możliwość oszacowania, jak daleko jest mu jeszcze do osiągnięcia celu, dla agenta używającego tych drugich takie szacowanie jest niedostępne. W rozdziale 4. uogólnimy naszą dyskusję na inne typy środowisk jednoagentowych, w rozdziale 5. zajmiemy się środowiskami wieloagentowymi.

W tym rozdziale wykorzystujemy intensywnie notację „dużego O", odzwierciedlającą asymptotyczną złożoność algorytmów — na przykład $O(n^2)$. Czytelników chcących uporządkować swą wiedzę na jej temat odsyłamy do Dodatku A.

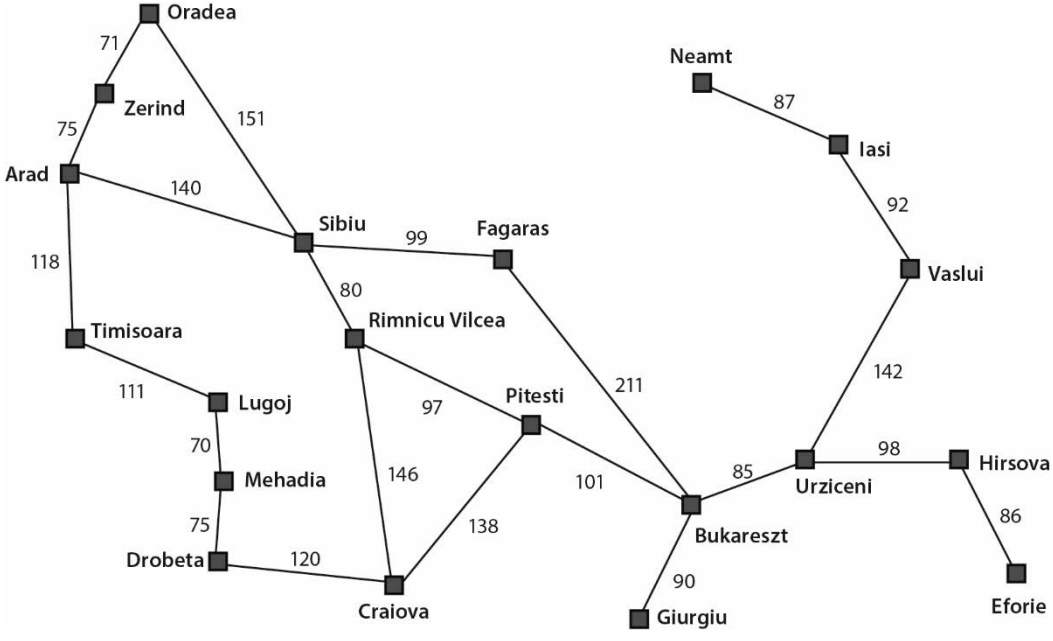
3.1. Agent rozwiązujący problem

Wyobraźmy sobie agenta, który spędza wakacje w Rumunii: zamierza zwiedzać zabytki, doskonalić znajomość języka, cieszyć się nocnym życiem (bez szkody dla samopoczucia następnego dnia) i tak dalej — wybór możliwości jest dość bogaty. Agent znajduje się obecnie w mieście Arad i ma bilet na lot z Bukaresztu następnego dnia. Według drogowskazów z Arad wychodzą trzy drogi: jedna w kierunku Sibiu, druga do Timishoary, trzecia do Zerind. Żadna z nich nie prowadzi bezpośrednio do Bukaresztu, więc podróżny, który nie zna geografii Rumunii, nie ma pomysłu na to, którą wybrać[1].

[1] Czytelników z Rumunii przepraszamy w tym miejscu za to, że omawiany przykład ma dla nich jakby mniejszy walor dydaktyczny.

Jeśli agent nie dysponuje jakimiś dodatkowymi informacjami — czyli jego środowisko jest dla niego nieznane — nie pozostaje mu nic lepszego, jak losowy wybór jednej z dróg; tę przykrą sytuację rozpatrujemy w rozdziale 4. Na użytek tego rozdziału zakładamy jednak, że nasi agenci zawsze dysponują informacjami o otaczającym świecie, chociażby takimi, jak mapa przedstawiona na rysunku 3.1. Nasz agent może tym samym rozwiązać swój problem, w czterech następujących etapach.

- **Zdefiniowanie celu.** Agent stawia sobie za główny cel dotarcie do Bukaresztu. Główny cel determinuje zachowanie agenta poprzez ograniczenie celów cząstkowych i tym samym akcji stojących do dyspozycji agenta.
- **Sformułowanie problemu.** Agent sporządza opis stanów i działań niezbędnych do osiągnięcia celu — czyli tworzy model części świata powiązanej z celem. Akcje w tym modelu ograniczają się do podróży z danego miasta do miast sąsiednich, zatem jedynym elementem stanu otaczającego świata, zmieniającym się wskutek wykonania akcji, jest bieżące położenie agenta.
- **Wyszukiwanie.** Zanim agent wykona jakąkolwiek akcję w rzeczywistym świecie, symuluje różne sekwencje akcji wynikające z jego modelu, wybierając ostatecznie tę sekwencję, która doprowadzi go do celu — sekwencję tę nazywamy **rozwiązaniem**. Oczywiście może się tak zdarzyć, że wśród sekwencji analizowanych przez agenta nie będzie żadnej, która mogłaby doprowadzić go do celu, w naszym przypadku jednak może to być (na przykład) sekwencja prowadząca wzdłuż trasy Arad – Sibiu – Fagars – Bukareszt.
- **Wykonanie.** Agent wykonuje teraz ciąg akcji, z których każda obejmuje jeden etap wspomnianej trasy.



RYSUNEK 3.1. Uproszczona mapa części Rumunii, z zaznaczonymi odległościami (w milach) między miastami

Zwróćmy uwagę na ważny fakt: agent, po wytyczeniu trasy, może ją przemierzać „z zamkniętymi oczami", czyli ignorować swe spostrzeżenia w tym czasie. Formalnie oznacza to, że *agent w czasie wykonywania swych akcji może ignorować zmiany zachodzące w środowisku, oczywiście jeśli zbudowany przez niego model jest poprawny; jest to cecha charakterystyczna dla środowiska rozpoznanego, deterministycznego i w pełni obserwowalnego, jeśli rozwiązaniem problemu jest ustalona sekwencja akcji*. Specjaliści od teorii sterowania nazywają taki stan systemem **otwartej pętli** (ang. *open loop*) — w czasie wykonywania sekwencji akcji pętla sprzężenia zwrotnego między agentem a środowiskiem zostaje przerwana („otwarta"). Jeśli istnieje prawdopodobieństwo, że model może być błędnie określony, bądź środowisko nie jest deterministyczne, agent byłby bezpieczniejszy, jeśli nie przerywałby obserwacji środowisk (co teoretycy sterowania nazywają **zamkniętą pętlą** — ang. *closed loop*), o czym piszemy w podrozdziale 4.4.

W środowiskach częściowo obserwowalnych lub niedeterministycznych rozwiązanie miałoby prawdopodobnie postać rozgałęzioną, zakładającą dynamiczny wybór kolejnego miasta na trasie w zależności od wyniku obserwacji środowiska. Przykładowo, agent znajdujący się w Arad może planować jazdę do Sibiu, ale bierze pod uwagę fakt, że może natrafić na zamknięty odcinek drogi (znak „droga zamknięta" — rum. *drum inchis*) bądź też (słabo orientując się na mapie) omyłkowo dojechać do Zerind, dlatego posiada plan awaryjny uwzględniający takie sytuacje.

3.1.1. Problemy wyszukiwania i ich rozwiązania

Problem wyszukiwania można formalnie zdefiniować jako kombinację następujących elementów:

- Zbiór możliwych **stanów**, w których może znajdować się środowisko; zbiór ten nazywamy **przestrzenią stanów**.
- **Stan początkowy**, czyli stan, w jakim znajduje się środowisko w momencie, gdy zaczynamy analizować działanie agenta; dla naszego agenta-podróżnika jest to *Arad*.
- **Stan docelowy** lub zbiór stanów docelowych. Dla naszego agenta-podróżnika jest to pojedynczy stan (*Bukareszt*), ogólnie jednak stany docelowe mogą tworzyć zbiór — być może niewielki, być może nawet nieskończony. Dla agenta-odkurzacza z rozdziału 2. stanem docelowym może być brak zabrudzenia na którymkolwiek kwadracie, niezależnie od innych aspektów stanu środowiska. Będziemy brać pod uwagę wszystkie trzy wymienione rodzaje stanu docelowego (pojedynczy stan, skończony zbiór stanów, nieskończenie wiele stanów) reprezentując je w postaci metody `IS_GOAL`. Dla prostoty będziemy w dalszej części rozdziału pisać po prostu „stan docelowy", lecz to określenie należy każdorazowo rozumieć jako „dowolny spośród możliwych stanów docelowych".
- **Akcje** dostępne dla agenta. Dla danego stanu s metoda `ACTIONS`(s) zwraca skończony[2] zbiór akcji, możliwych do wykonania w stanie s. O każdej z tych akcji mówimy, że jest **stosowalna** do stanu s. Na przykład

  `ACTIONS(`*Arad*`) = {`*ToSibiu*`, `*ToTimisoara*`, `*ToZerind*`}`

- **Model przejścia** określa skutki wykonywania akcji — wynikiem wywołania metody `RESULT`(s, a) jest stan będący rezultatem wykonania akcji a w stanie s, na przykład

  `RESULT(`*Arad*`, `*ToZerind*`) = `*Zerind*

- **Funkcja kosztu akcji**, oznaczana przez `ACTION-CONST`(s, a, s') lub $c(s, a, s')$ oznacza wyrażony liczbowo koszt osiągnięcia stanu s' jako skutku wykonania akcji a w stanie s. Agent rozwiązujący problemy powinien wykorzystywać tę funkcję jako odzwierciedlającą jego własną miarę wydajności, w przypadku naszego agenta-podróżnika kosztem przejścia ze stanu s do stanu s' może być odległość między miastami odpowiadającymi tym stanom, według mapy z rysunku 3.1, albo czas przejazdu między tymi miastami.

[2] Do rozwiązywania problemów, w których stanom odpowiadają nieskończone zbiory akcji, konieczne jest zastosowanie technik wykraczających poza zakres tego rozdziału.

Sekwencja działań tworzy **ścieżkę**, a **rozwiązanie** problemu to ścieżka prowadząca od stanu początkowego do stanu docelowego. Zakładamy, że koszty akcji są *addytywne*; czyli że całkowity koszt ścieżki jest sumą kosztów akcji tworzących tę ścieżkę. **Rozwiązaniem optymalnym** wśród wszystkich możliwych rozwiązań jest to, któremu odpowiada ścieżka o najmniejszym koszcie. W tym rozdziale, dla uniknięcia zbytnich komplikacji, zakładamy, że wszystkie koszty akcji są dodatnie[3].

Przestrzeń stanów można przedstawić w postaci **grafu**, którego wierzchołki reprezentują stany, a skierowane krawędzie między nimi są odpowiednikami akcji.

Mapa przedstawiona na rysunku 3.1 jest grafem, którego krawędzie nie są skierowane. Krawędź nieskierowaną można traktować jako parę krawędzi skierowanych przeciwnie względem siebie, zatem każdą drogę na wspomnianej mapie można traktować jako parę akcji związanych z przemieszczeniami między miastami określonej pary.

3.1.2. Formułowanie problemów

Nasze sformułowanie problemu dotarcia z Arad do Bukaresztu to w istocie **model** — abstrakcyjny opis matematyczny, jedynie przybliżający rzeczywistość. W porównaniu z licznymi aspektami podróży po kraju — towarzyszami podróży, dostępnością ulubionych stacji radiowych, krajobrazami za oknem, bliskością patroli policyjnych, odległością między parkingami, stanem dróg, warunkami pogodowymi — model ogranicza się wyłącznie do ścieżki prowadzącej od stanu *Arad* do stanu *Bukareszt*, reszta nie ma znaczenia dla rozwiązania problemu.

Takie ignorowanie nieistotnych szczegółów nosi nazwę **abstrakcji**. Dobrze zbudowany model charakteryzuje się odpowiednim poziomem abstrakcji: gdybyśmy akcje agenta-podróżnika opisywali na poziomie szczegółowości typu „przesuń prawą stopę o centymetr do przodu" czy „obróć kierownicę o jeden stopień w kierunku przeciwnym do ruchu wskazówek zegara", agent nie tylko nie zdążyłby na samolot w Bukareszcie, ale mnóstwo czasu zajęłoby mu w ogóle wyjechanie z parkingu.

Żeby być bardziej precyzyjnymi w kwestii **poziomu abstrakcji**: zwróć uwagę, że wybrane przez nas stany i akcje uosabiają duże zbiory szczegółowych stanów świata rzeczywistego i szczegółowych sekwencji działań. Rozważmy nieco inną trasę naszego agenta — Arad – Sibiu – Rimnicu Vilcea – Pitesti – Bukareszt — i nieco bardziej szczegółowo określoną sekwencję akcji: na odcinku od Sibiu do Rimnicu Vilcea agent słucha radia, potem jednak siła sygnału jego ulubionej stacji szybko zanika, więc agent wyłącza radio na resztę podróży.

Abstrakcja jest *poprawna*, jeśli możemy przy jej użyciu uzyskać dowolne abstrakcyjne rozwiązanie odpowiadające bardziej szczegółowemu rozwiązaniu w bardziej szczegółowym świecie, pod jednym wszakże warunkiem — każdej abstrakcyjnej akcji (na przykład *ToSibiu*) odpowiada sekwencja bardziej szczegółowych akcji, powodująca identyczną zmianę między tymi samymi stanami (na przykład *WyjeżdżamZParkinguWArad*, *UzupełniamPaliwo*, *JadęWKierunkuAutostrady*, ..., *ZjeżdżamZAutostrady*, *WjeżdżamNaParkingWSibiu*)[4]. Abstrakcja jest *użyteczna*, jeśli każda z akcji

[3] Jeśli dopuścimy ujemną wartość kosztu akcji, to możliwe będzie powstawanie cykli o zerowym lub ujemnym koszcie. Optymalnym rozwiązaniem problemu zawierającego cykl o ujemnym koszcie netto jest przebycie tego cyklu nieskończoną liczbę razy. Algorytmy Bellmana-Forda i Floyda-Warshalla (nie omawiamy ich tutaj) radzą sobie z ujemnymi kosztami (wagami krawędzi), o ile w grafie nie występują (właśnie) cykle o ujemnych kosztach netto. Jeśli chodzi o akcje o *zerowym* koszcie, to łatwo je uwzględniać pod warunkiem, że liczba takich akcji następujących bezpośrednio po sobie jest a priori ograniczona. Możemy na przykład dysponować robotem, którego każdy ruch postępowy wiąże się z określonym kosztem, ale koszt obrotu o kąt prosty jest zerowy: algorytmy omawiane w tym rozdziale radzą sobie z takim przypadkiem, o ile dozwolone są co najwyżej trzy takie obroty „z rzędu", czyli nieprzedzielone ruchem postępowym. Kolejnym trudnym przypadkiem są problemy zawierające ścieżki o skończonym koszcie, jednak złożone z *nieskończonej* liczby akcji o małych kosztach — zjawisko to jest istotą paradoksu Zenona, polegającego na tym, że analizujemy jednostajny ruch żółwia, dzieląc odmierzany czas na odcinki, z których każdy równy jest połowie poprzedniego. Aby żółw mógł dotrzeć do celu, potrzebna jest nieskończona liczba takich odcinków, czyli nieskończony czas, ergo żółw nigdy do celu nie dotrze (od dawna wiadomo jednak, gdzie tkwi błąd w tym rozumowaniu). By uniknąć takich subtelnych problemów wymagamy, by koszt każdej akcji był nie mniejszy niż pewien ustalony limit $\varepsilon > 0$.

[4] Patrz podrozdział 11.4.

prowadzących do abstrakcyjnego rozwiązania jest prostsza niż odpowiadająca jej sekwencja akcji w świecie rzeczywistym; w naszym przykładzie akcja *ToSibiu*, oznaczająca „przejedź z Arad do Sibiu" bez sprecyzowania jakichkolwiek dalszych szczegółów, jest jednoznacznie zrozumiała dla kierowcy o przeciętnych umiejętnościach. Opracowanie dobrej abstrakcji sprowadza się zatem do wyeliminowania jak największej liczby nieistotnych szczegółów, oczywiście przy zachowaniu jej poprawności i zapewnieniu, że każda abstrakcyjnie sformułowana akcja da się prosto przełożyć na konkretne działania w rzeczywistości. Gdyby nie umiejętność budowania użytecznych abstrakcji przez projektantów, każdy inteligentny agent rychło utonąłby w rzeczywistym środowisku.

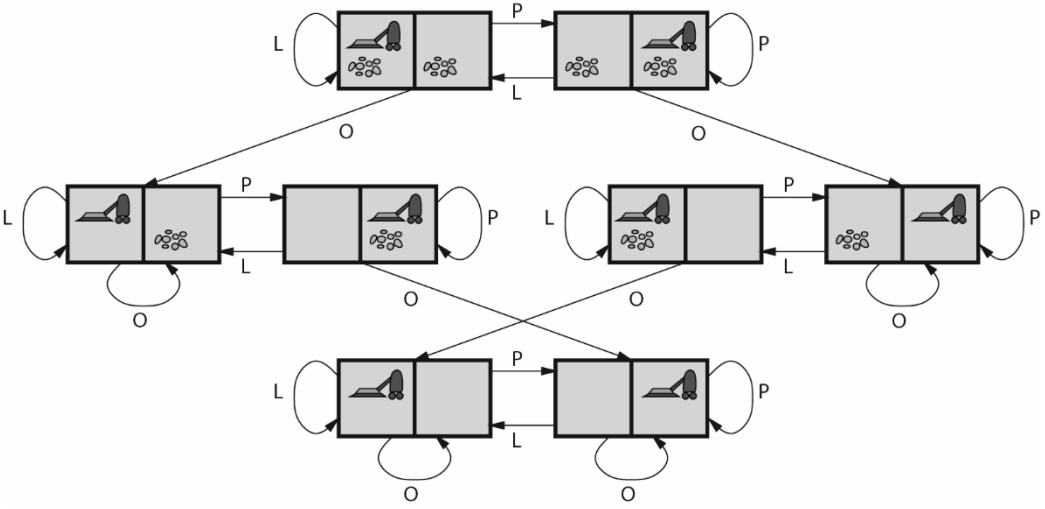
3.2. Przykładowe problemy

Opisane podejście do rozwiązywania problemów daje się zastosować do szerokiej gamy różnych środowisk zadaniowych, zaprezentujemy je w odniesieniu do środowisk najbardziej znanych. Będziemy przy tym odróżniać problemy *rzeczywiste* od problemów *standaryzowanych*. **Problemy standaryzowane** służą do ilustracji i nauki określonych metod poszukiwania rozwiązań; charakteryzują się zwięzłym, dokładnym opisem i z tego względu mogą służyć jako punkty odniesienia dla porównywania wydajności różnych algorytmów. **Problemy rzeczywiste** — takie jak nawigowanie robotów — to takie, których rozwiązania są bezpośrednio wykorzystywane przez ludzi; a sformułowanie każdego z nich jest specyficzne, dalekie od normalizacji. Przyczyną tej specyfiki jest zróżnicowanie szczegółów środowiska zadaniowego, na przykład różne roboty używają różnego zestawu czujników, odbierając w efekcie różne ciągi percepcyjne.

3.2.1. Problemy standaryzowane

Środowiskiem **problemu gridowego** jest dwuwymiarowa prostokątna tablica (macierz) komórek, często określana przejętym z języka angielskiego terminem *grid*. Agent może poruszać się po tej tablicy w poziomie, w pionie i (w przypadku niektórych instancji problemu) na skos, *o ile nie istnieje przeszkoda uniemożliwiająca przemieszczenie się do sąsiedniej komórki*. Komórki mogą zawierać obiekty, które agent może podnosić, pchać, ciągnąć i w jeszcze inny sposób na nie oddziaływać; dwie sąsiednie komórki mogą być oddzielone ścianą, która uniemożliwia bezpośrednie przemieszczenie obiektu między nimi. Typową odmianą problemu gridowego jest problem agenta-odkurzacza z rozdziału 2., który to problem można sformułować następująco — zgodnie z przedstawioną wcześniej definicją:

- **Stany.** Jedynymi obiektami, które mogą znajdować się w komórkach, są: agent i kurz. Każda z komórek może znajdować się w dwóch stanach: *Czysta* albo *Brudna*, w oryginalnej wersji problemu macierz zawiera dwie komórki, co daje 2×2 = 4 stany zabrudzenia. Niezależnie od tego odkurzacz może znajdować się w jednej z dwóch komórek, co daje ogólną liczbę stanów środowiska 4×2 = 8 (jak pokazano na rysunku 3.2). Ogólnie środowisko n-komórkowe może znajdować się w jednym z $n \times 2^n$ stanów.
- **Stan początkowy.** Dowolny stan może pełnić rolę stanu początkowego.
- **Akcje.** W dwukomórkowym świecie odkurzacza zdefiniowaliśmy trzy akcje: *Odkurz*, *W_lewo* i *W_prawo*, w dwuwymiarowym świecie wielokomórkowym potrzebowalibyśmy dodatkowych akcji ruchowych, przede wszystkim *W_górę* i *W_dół*, co dałoby nam cztery **absolutne** rodzaje ruchów. Alternatywą mogłyby być cztery ruchy **egocentryczne**, czyli relatywne do bieżącej pozycji odkurzacza: *W_przód*, *W_tył Skręć_w_lewo*, *Skręć_w_prawo*.



RYSUNEK 3.2. Graf przestrzeni stanów dla dwukomórkowego środowiska odkurzacza, uwidoczniający 8 stanów i trzy akcje: L — *W_lewo*, P — *W_prawo*, O — *Odkurz*

- **Model przejścia.** Akcja *Odkurz* usuwa wszelkie zabrudzenia z komórki, w której aktualnie znajduje się odkurzacz. Akcja *W_przód* przesuwa odkurzacz do sąsiedniej komórki w kierunku, w którym jest zwrócony swym przodem, wyjątek stanowi sytuacja, gdy przed odkurzaczem znajduje się ściana i wspomniany ruch nie jest możliwy, wówczas akcja nie wywołuje żadnego ruchu. Analogicznie, akcja *W_tył* powoduje ruch w przeciwnym kierunku, o ile takowy jest możliwy. Akcja *W_lewo* powoduje obrót odkurzacza o kąt prosty w kierunku antyzegarowym, analogicznie akcja *W_prawo* powoduje obrót odkurzacza o kąt prosty w kierunku zegarowym; w obu przypadkach odkurzacz nie opuszcza bieżącej komórki.
- **Stany docelowe.** Stanami docelowymi są wszystkie stany, w których żadna z komórek nie zawiera brudu.
- **Koszt akcji.** Koszt każdej akcji równy jest 1.

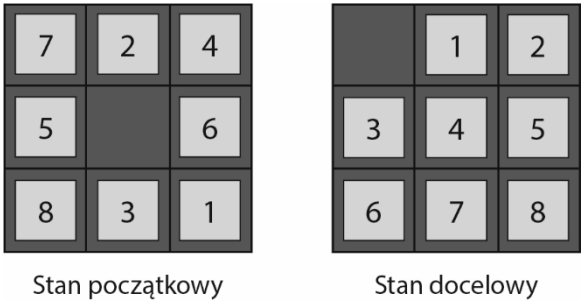
Środowiskiem innego problemu gridowego jest odmiana japońskiej układanki **Sokoban**. Każda z komórek gridu albo jest pusta, albo zawiera jedną skrzynię. „Rozwiązanie" układanki polega na przemieszczeniu każdej skrzyni do wyznaczonej dla niej a priori komórki docelowej. Gdy agent wchodzi do komórki zajętej przez skrzynię, a któraś z sąsiednich komórek jest pusta i nie jest oddzielona ścianą, agent wraz ze skrzynią przemieszcza się do tej komórki. Agentowi nie wolno przepychać skrzyni do innej komórki. W środowisku z n komórkami i b skrzyniami, bez odgradzających ścian, liczba możliwych stanów wynosi

$$\frac{n \times n!}{b! \times (n-b)!}$$

co dla gridu 8×8 komórek i 12 skrzyń daje 21×10^{13}.

Układanka doczekała się kilku znanych wariantów. W wariancie **przesuwnych kafelków** (ang. *sliding-tile puzzle*) w gridzie istnieje co najmniej jedna wolna komórka, na którą można przesuwać sąsiadujące kafelki. W wariancie o nazwie *Rush Hour* („godziny szczytu") w komórkach gridu 6×6 znajdują się pojazdy — 12 samochodów osobowych i 4 ciężarówki; samochód osobowy ma wymiary 1×2 komórki, ciężarówka 1×3 komórki; zadaniem gracza jest uwolnienie

wskazanego pojazdu z „korka" ulicznego. Najbardziej chyba znanymi odmianami układanki są tzw. **puzzle-8**[5] — 8 kafelków przesuwalnych wewnątrz gridu 3×3 komórki (rysunek 3.3) oraz **puzzle-15** — 15 kafelków w gridzie 4×4 komórki. Celem gry jest „uporządkowanie kafelków", czyli sprawienie, by każdy z nich znalazł się na swoim miejscu (jak w prawej części rysunku 3.3).



RYSUNEK 3.3. Przykładowa instancja gry puzzle-8

Standaryzowane sformułowanie problemu dla odmiany puzzle-8 prezentuje się następująco:

- **Stany.** Stan to położenia poszczególnych kafelków.
- **Stan początkowy.** Dowolny stan może pełnić rolę stanu początkowego. Zauważmy, że własność parzystości dzieli przestrzeń stanów na pół — dokładnie połowa wszystkich stanów początkowych umożliwia osiągnięcie danego stanu docelowego. (patrz Ćwiczenie online 3. PART).
- **Akcje.** W świecie rzeczywistym akcję stanowi oczywiście przesunięcie kafelka, lecz ze względu na prostotę opisu wygodniej będzie rozpatrywać manipulowanie układanką jako przesuwanie się pustej komórki *W_lewo*, *W_prawo*, *W_górę* i *W_dół*. Oczywiście gdy pusta komórka znajduje się w którymś ze skrajnych położeń, nie wszystkie z tych akcji mają zastosowanie.
- **Model przejścia.** Mapuje parę (stan bieżący, akcja) w nowy stan bieżący, na przykład zastosowanie akcji *W_lewo* do stanu widocznego w lewej części rysunku 3.3 spowoduje zamianę kafelka nr 5 z pustą komórką.
- **Stan docelowy.** Chociaż możemy wybrać dowolny stan w charakterze stanu docelowego, to zwykle przyjmuje się w tej roli uporządkowanie kafelków według numerów, jak w prawej części rysunku 3.3.
- **Koszt akcji.** Koszt każdej akcji równy jest 1.

Zauważmy, że każde sformułowanie problemu zawiera abstrakcje. Definicja każdej z akcji układanki puzzle-8 obejmuje wyłącznie układ kafelków przed i po wykonaniu akcji — pomijamy możliwość zablokowania się kafelka w trakcie przesuwania (i potrząsania tabliczką w celu likwidacji blokady), nie dopuszczamy także „wydłubywania" kafelków z tabliczki (na przykład za pomocą noża) i wstawiania ich na nowe miejsce. Uwzględnienie takich i podobnych fizycznych manipulacji niepotrzebnie zaciemniłoby obraz problemu.

[5] W amerykańskiej terminologii układanka z *k* kafelkami określana jest terminem „*k*-puzzle", w polskim wydaniu tej książki konsekwentnie będę używał sformułowań „puzzle-8", puzzle-15" i „puzzle-24" dla układanki z (odpowiednio) 8, 15 i 24 kafelkami poruszającymi się na kwadratowej planszy o boku (odpowiednio) 3, 4 i 5 — *przyp. tłum.*

Ostatni z rozpatrywanych przez nas problemów zestandaryzowanych pochodzi od Donalda Knutha (1964) i ilustruje pojawienie się nieskończonej przestrzeni stanów. Knuth wyraził mianowicie przypuszczenie, że wychodząc od liczby 4 i stosując odpowiednio funkcje pierwiastka kwadratowego, obcięcia w dół do liczby całkowitej (ang. *floor*) i silni, możemy otrzymać jako wynik dowolną liczbę naturalną — na przykład 5:

$$floor\left(\sqrt{\sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}}}\right) = 5$$

Definicja tego problemu jest nadzwyczaj prosta:

- **Stany.** Dodatnie liczby rzeczywiste.
- **Stan początkowy.** 4.
- **Akcje.** Użycie funkcji $\sqrt{}$, *floor* lub silni (ta ostatnia tylko do argumentów będących liczbami naturalnymi).
- **Model przejścia.** Wynika z definicji użytych funkcji matematycznych.
- **Stan docelowy.** Żądana liczba naturalna.
- **Koszt akcji.** Koszt każdej akcji równy jest 1.

Przestrzeń stanów dla tego problemu jest nieskończona — silnia liczby naturalnej większej niż 2 jest od tej liczby większa. Stany problemu mogą mieć postać bardzo dużych liczb: w przedstawionym przykładzie wychodzimy od 4 i w pierwszym kroku obliczamy 4! = 24, jednak już w drugim kroku mamy (bagatela)

24! = 620 448 401 733 239 439 360 000

Nieskończone przestrzenie stanów pojawiają się często w zadaniach obejmujących generowanie wyrażeń matematycznych, dowodów twierdzeń, programów, obwodów logicznych i innych obiektów definiowanych rekurencyjnie.

3.2.2. Problemy ze świata rzeczywistego

Widzieliśmy już, jak *problem ze znajdowaniem trasy* jest definiowany w kategoriach określonych lokalizacji i przemieszczania się po łączących je drogach. Podobne algorytmy są używane w wielu różnych aplikacjach. Niektóre, takie jak systemy (webowe lub wbudowane w pokładowe komputery samochodów) prostego wyznaczania trasy na podstawie wyłącznie sieci dróg, stanowią proste rozszerzenia naszego rumuńskiego przykładu; ograniczenie ich użyteczności jest konsekwencją ignorowania zróżnicowanych kosztów przejazdu przez poszczególne drogi — zróżnicowanych z powodu różnego natężenia ruchu czy też zamknięcia pewnych odcinków z powodu robót drogowych. Aplikacje bardziej skomplikowane, służące (między innymi) do trasowania strumieni wideo w sieciach komputerowych, planowania operacji wojskowych czy układania rozkładów lotów, charakteryzują się znacznie większą złożonością.

Rozważmy problem projektowania serwisu webowego ułatwiającego planowanie podróży lotniczych — poniższa specyfikacja, i tak mocno uproszczona, jest o wiele bardziej złożona niż ta z rumuńskiego przykładu.

- **Stany.** Oczywistymi składnikami stanu są: bieżąca lokalizacja pasażera i lokalny czas. Ponadto, jako że koszt danego odcinka podróży może zależeć od kosztów odcinków poprzednich, ich taryf przewozowych i statusów (krajowy albo międzynarodowy), stan musi zawierać informacje o dotychczasowej historii podróży.
- **Stan początkowy.** Port lotniczy, z którego pasażer rozpoczyna podróż.
- **Akcje.** Wybierz dowolny lot z bieżącej lokalizacji, w dowolnej klasie miejsc, rozpoczynający się po bieżącej godzinie, z uwzględnieniem czasu niezbędnego na przesiadkę i formalności na lotnisku.

- **Model przejścia.** Parametrami nowego stanu, po przebyciu odcinka podróży, są między innymi: port końcowy tego odcinka i obowiązujący w tym porcie lokalny czas.
- **Stan docelowy.** Port końcowy ostatniego odcinka. Niekiedy cel może być bardziej skomplikowany, na przykład może uwzględniać wymaganie podróży bez międzylądowań.
- **Koszt akcji.** Kombinacja różnych czynników, między innymi kosztu pieniężnego, całkowitego czasu podróży i czasu oczekiwania przy przesiadkach, klasy miejsc, formalności celnych i imigracyjnych, pory dnia, typu samolotu i punktów lojalnościowych dla osób często podróżujących.

W przypadku komercyjnych systemów wspomagających planowanie podróży specyfikacja obejmuje wiele innych elementów, wynikających z (na przykład) bizantyjsko zagmatwanych systemów taryf. Poza tym doświadczony pasażer znakomicie zdaje sobie sprawę z faktu, że samoloty niekoniecznie kursują zgodnie z planem; dobry system planowania podróży powinien zatem przewidywać plan awaryjny na wypadek, gdy opóźnienie lub odwołanie lotu zniweczy zaplanowany rozkład podróży.

Problemy turystyczne tym się różnią od opisywanego problemu planowania podróży, że ich celem jest nie pojedyncza, końcowa lokalizacja, lecz *zbiór lokalizacji*, które mają zostać w ramach wycieczki odwiedzone. Jednym z takich problemów jest **problem komiwojażera** (ang. TSP — *traveling salesperson problem*)**:** zadaniem komiwojażera (akwizytora, sprzedawcy, przedstawiciela handlowego) jest odwiedzenie wszystkich miast z ustalonego zbioru, oczywiście zgodnie z istniejącym systemem połączeń (dróg) między miastami. Każde połączenie między miastami cechuje się określonym kosztem, a celem komiwojażera jest wybór takiej kolejności odwiedzania miast (czyli takiej trasy), by łączny koszt wszystkich przebytych dróg był mniejszy od ustalonej wartości *C* (w optymalizacyjnej wersji problemu celem komiwojażera jest wybór trasy o *najmniejszym możliwym* koszcie). Informatycy włożyli mnóstwo wysiłków w badania i rozwój algorytmów rozwiązujących problem komiwojażera, ponieważ algorytmy te bardzo łatwo dają się przystosowywać do problemu zarządzania flotą pojazdów — ich zastosowanie do optymalizacji wykorzystania autobusów szkolnych w Bostonie przyniosło oszczędności rzędu 5 milionów USD, a ponadto zmniejszenie zanieczyszczenia środowiska, zmniejszenie natężenia ruchu oraz oszczędność czasu uczniów i kierowców (Bertsimas i in., 2019). Pokrewne algorytmy znalazły zastosowanie do planowania ruchów automatycznych wierteł do robienia otworów w płytkach drukowanych i optymalizacji wykorzystania wózków widłowych w magazynach.

Układy wielkiej skali integracji (ang. VLSI — *Very Large Scale of Integration*) to miliony tranzystorów i połączeń między nimi w pojedynczym chipie, które należy rozmieścić w taki sposób, by zminimalizować wymaganą powierzchnię, opóźnienia sygnałów i pojemności rozproszone, a jednocześnie zmaksymalizować wydajność produkcji. Problem **topografii** chipu pojawia się po zakończeniu fazy projektowania logicznego i rozwiązywany jest zwykle w dwóch fazach: projektowania **układu komórek** i **trasowania kanałów**. W pierwszej z tych faz pierwotne komponenty pogrupowane zostają w komórki, z których każda spełnia dobrze określoną funkcję, charakteryzuje się określonym kształtem i rozmiarem oraz wymaga określonej liczby połączeń z każdą z pozostałych komórek. Oczywiście komórki muszą zajmować rozłączne obszary (nie mogą zachodzić na siebie), a po ich rozmieszczeniu musi pozostać wystarczająco dużo miejsca na łączące je przewody. Projektowanie określonego *przebiegu* wspomnianych przewodów to właśnie trasowanie kanałów. Dla projektantów są to zadania ekstremalnie skomplikowane, ale warte rozwiązywania, i — o czym świadczą coraz to nowsze generacje procesorów — rozwiązywalne.

Nawigowanie robotów to uogólnienie opisanego wcześniej problemu znajdowania trasy: zamiast podążać ustalonymi ścieżkami (jak w naszym rumuńskim przykładzie), robot może wędrować po okolicy, wytyczając sobie własne ścieżki. Z perspektywy robota krążącego po płaskiej powierzchni problem ten jest w zasadzie dwuwymiarowy, ale już wyposażenie robota w (sterowalne) ramiona i nogi przydaje problemowi dodatkowych wymiarów, po jednym na każdy staw i przegub kończyny. Komplikuje to ów problem na tyle, że już samo zapewnienie *skończoności* przestrzeni poszukiwań, geometrycznie ciągłej przecież, wymaga zaawansowanych technik (patrz rozdział 26.). A to dopiero początek złożoności: funkcjonujący w praktyce robot musi dodatkowo radzić sobie z błędami w odczytach czujników i sterowaniu silnikami, z częściową obserwowalnością otoczenia i innymi agentami, którzy mogą wpływać na stan otoczenia.

Automatyczne sekwencjonowanie montażu złożonych obiektów (takich jak silniki elektryczne) przez roboty jest standardową praktyką przemysłową stosowaną od niemal półwiecza — minimalizowanie ręcznej pracy ludzkiej na liniach montażowych przynosi zwykle znaczne oszczędności czasu i kosztów. Algorytm sekwencjonowania najpierw znajduje jakąś wykonalną sekwencję montażu, po czym dąży do jej optymalizacji; jeżeli wybierze sekwencję niewykonalną, na którymś jej etapie okaże się, że dodanie kolejnej części jest niemożliwe bez cofnięcia (czyli zmarnotrawienia) części już wykonanej pracy. Badanie wykonalności sekwencji działań stanowi trudny problem wyszukiwania geometrycznego, ściśle powiązany z problemem nawigowania robotów, zatem generowanie kolejnych dopuszczalnych akcji jest najdroższą częścią procesu sekwencjonowania. Przeszukiwanie kompletnej przestrzeni możliwych akcji jest zadaniem niewykonalnym obliczeniowo, więc każdy praktyczny algorytm sekwencjonowania musi ograniczać się do niewielkiej części tej przestrzeni.

Pokrewnym sekwencjonowaniu czynności montażowych jest problem **projektowania białek**. Jego celem jest znajdowanie sekwencji aminokwasów, które zwijać (fałdować) się będą w trójwymiarowe struktury białek, posiadających własności leczenia niektórych chorób.

3.3. Algorytmy wyszukiwania

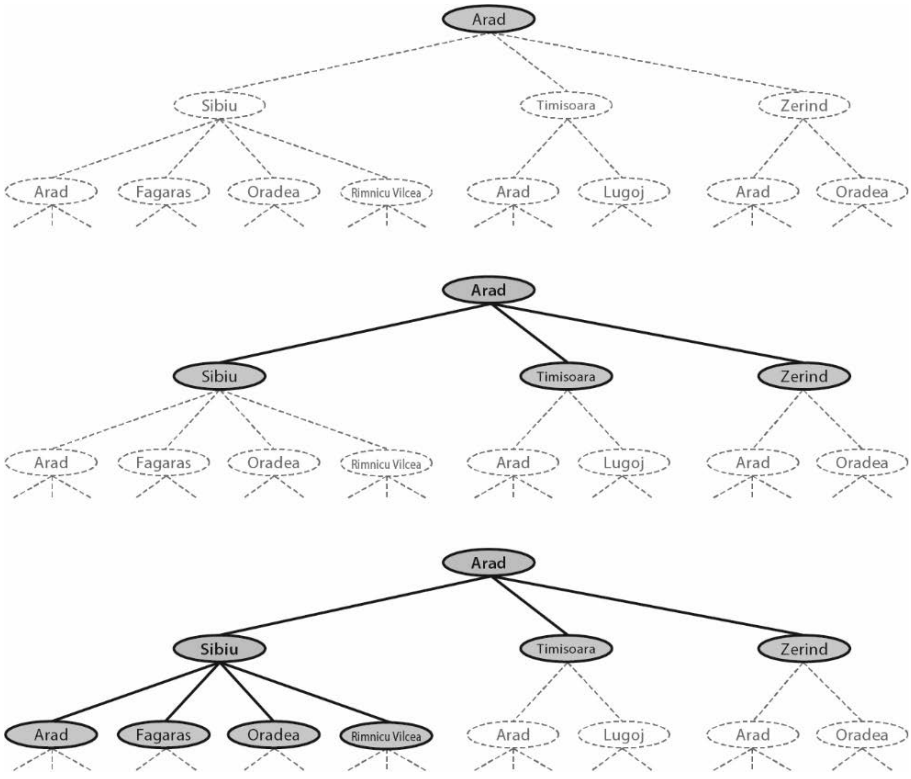
Wejściem do **algorytmu wyszukiwania** (ang. *search algorithm*) jest problem do rozwiązania, a wynikiem — rozwiązanie problemu albo informacja, że takie rozwiązanie nie istnieje. W tym rozdziale opisujemy algorytmy, które nakładają **drzewo wyszukiwawcze** na graf przestrzeni stanów; drzewo to zawiera różne ścieżki rozpoczynające się od stanu początkowego, a algorytm próbuje odnaleźć ścieżkę, która kończy się w stanie docelowym. Każdy **węzeł** (ang. *node*) tego drzewa reprezentuje jeden stan z przestrzeni stanów, a każda **gałąź** (ang. *branch*) reprezentuje akcję powodującą przejście między stanami. Węzeł stanowiący wspólny początek wszystkich ścieżek, odpowiadający stanowi początkowemu, nazywamy **korzeniem** (ang. *root*).

Należy wyraźnie odróżniać drzewo wyszukiwawcze od grafu przestrzeni stanów. W grafie tym każdy stan z przestrzeni (być może nieskończonej) odwzorowywany jest w jeden wierzchołek, a każda krawędź reprezentuje wykonanie określonej akcji w określonym stanie. W drzewie wyszukiwawczym kolejne węzły na danej ścieżce reprezentują pewien ciąg stanów; w drzewie tym może istnieć wiele ścieżek, a w konsekwencji wiele węzłów odpowiadających danemu stanowi. Drzewo z definicji nie zawiera cykli, zatem każdemu węzłowi odpowiada dokładnie jedna ścieżka prowadząca wstecz do korzenia.

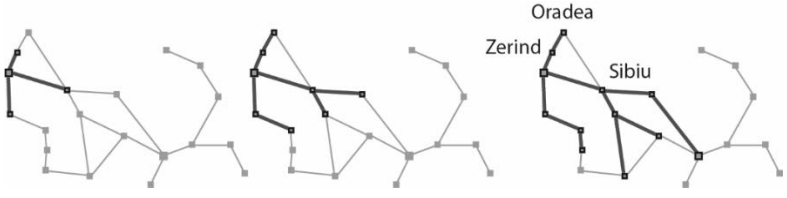
Rysunek 3.4 przedstawia część drzewa wyszukiwawczego, reprezentującą kilka pierwszych kroków w poszukiwaniu trasy z Arad do Bukaresztu. Korzeń odpowiada stanowi głównemu, czyli miejscowości *Arad*; uwzględniając akcje (`ACTIONS`) dostępne w tym stanie możemy węzeł **rozszerzyć**, czyli za pomocą funkcji `RESULT`, zobaczyć dokąd prowadzą te akcje i dla każdej z nich **wygenerować** nowy węzeł, zwany **węzłem potomnym** (ang. *child node*) lub **następnikiem** (ang. *successor*) korzenia, który dla węzła potomnego jest **węzłem macierzystym** (ang. *parent node*) lub **poprzednikiem** (ang. *precedessor*) — operacja ta nosi nazwę **rozwinięcia** (ang. *expansion*) węzła macierzystego.

Teraz musimy wybrać, który z tych trzech węzłów potomnych rozważymy jako następny — to właśnie stanowi istotę wyszukiwania: podążanie za jedną wybraną opcją i pozostawianie innych na później. Wybierzmy zatem jako pierwszy do rozwinięcia węzeł *Sibiu*. Rezultatem tego rozwinięcia (patrz dolna część rysunku 3.4) jest wygenerowanie 6 nierozwiniętych węzłów (etykietowanych pogrubioną czcionką). Zbiór ten nazywamy **granicą** (ang. *frontier*). Stan, dla którego został wygenerowany węzeł, nazywamy stanem **osiągniętym** (ang. *reached*) niezależnie od tego, czy węzeł ten został rozwinięty, czy nie[6],a węzeł reprezentujący ten stan — **węzłem osiągniętym.** Kolejne stadia rozwojowe drzewa wyszukiwawczego nałożonego na graf przestrzeni stanów widoczne są na rysunku 3.5.

[6] Niektórzy autorzy nazywają granicę drzewa **otwartą listą** (ang. *open list*), co jest zarówno mniej sugestywne pod względem geograficznym, jak i mniej adekwatne z perspektywy obliczeń — kolejka (ang. *queue*) jest w tej sytuacji bardziej efektywna niż lista. Konsekwentnie zbiór węzłów wcześniej rozwiniętych nazywany jest **listą zamkniętą** (ang. *closed list*), co w naszym ujęciu równoważne jest zbiorowi węzłów osiągniętych pomniejszonemu o granicę.

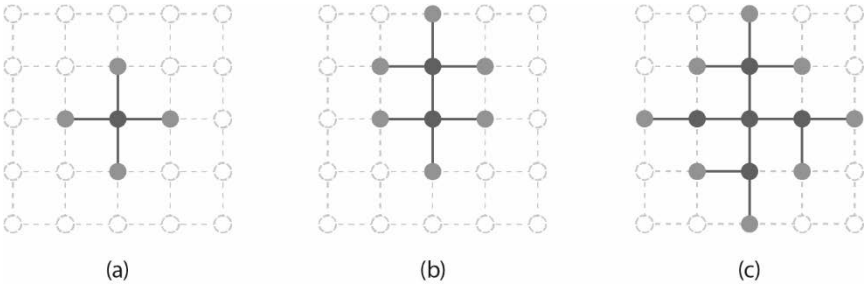


RYSUNEK 3.4. Trzy częściowe drzewa wyszukiwawcze odpowiadające poszukiwaniu drogi z Arad do Bukaresztu. Węzły rozwinięte mają kolor fioletowy i pogrubioną czcionkę w etykiecie, węzły graniczne, wygenerowane, ale jeszcze nie rozwinięte, mają kolor zielony i etykietowane są przy użyciu zwykłej czcionki. Stany odpowiadające obu tym rodzajom węzłów nazywane są stanami osiągniętymi. Węzły, które mogą zostać wygenerowane jako kolejne, otoczone są cienką linią przerywaną. Zauważmy, że w dolnym drzewie występuje cykliczne następstwo stanów Arad – Sibiu – Arad, wykluczające optymalność ścieżki, zatem ścieżka ta nie powinna być dalej rozwijana



RYSUNEK 3.5. Sekwencja wybranych etapów rozwojowych drzewa wyszukiwawczego generowanego przez graf z rysunku 3.1. Na każdym etapie rozwijamy wszystkie węzły tworzące granicę, rozszerzając w ten sposób ścieżki, ograniczając się do akcji, które mają sens i nie prowadzą do któregoś ze stanów już osiągniętych. Zauważmy, że w trzecim etapie rozwinięcie węzła Oradea doprowadziłoby do wygenerowania dwóch węzłów potomnych, reprezentujących stany już osiągnięte (Zerind i Sibiu), dlatego rezygnujemy z tego rozwinięcia

Zauważmy, że granica **oddziela** w grafie przestrzeni stanów dwa regiony: wewnętrzny, zawierający węzły już rozwinięte, i zewnętrzny, zawierający węzły reprezentujące stany, które nie zostały jeszcze osiągnięte. Właściwość tę zilustrowaliśmy na rysunku 3.6.



RYSUNEK 3.6. Zjawisko separacji przy przeszukiwaniu grafu, zilustrowane za pomocą prostokątnej siatki (gridu). Granica (kolor zielony) oddziela region wewnętrzny (kolor fioletowy) od zewnętrznego (brak koloru). Granica stanowi zbiór węzłów reprezentujących stany osiągnięte, ale jeszcze nierozwiniętych. Region wewnętrzny to zbiór węzłów już rozwiniętych, węzły w regionie zewnętrznym reprezentują natomiast stany jeszcze nieosiągnięte. W stadium (a) rozwinięty jest jedynie korzeń drzewa, w stadium (b) rozwinięty jest najwyżej położony węzeł granicy, stadium (c) to efekt rozwinięcia pozostałych węzłów potomnych korzenia, odwiedzanych w kolejności zgodnej z ruchem wskazówek zegara

3.3.1. Wyszukiwanie „najpierw najlepszy"

Skoro rozbudowa drzewa wyszukiwawczego odbywa się poprzez rozwijanie węzłów tworzących granicę, to naturalną kwestią staje się *kolejność* wyboru poszczególnych węzłów do rozwinięcia. Ogólna strategia rozwiązywania tej kwestii nazywana jest potocznie „**najpierw najlepszy**" (ang. *best first*) i wyznacza tę kolejność na podstawie pewnej **funkcji ewaluacyjnej** (ang. *evaluation function*) — im *mniejsza* wartość tej funkcji dla konkretnego węzła, tym wcześniej węzeł ten zostanie rozwinięty. W programie na listingu 3.1 funkcja ta jest parametrem o nazwie *f*. W każdej iteracji wybieramy, spośród nierozwiniętych jeszcze węzłów tworzących granicę, ten węzeł *n*, dla którego wartość $f(n)$ jest najmniejsza; jeżeli węzeł ten reprezentuje stan docelowy, zwracamy go jako wynik, w przeciwnym razie wywołujemy dla tego węzła funkcję `EXPAND` generującą węzły potomne. Jeżeli dany węzeł potomny reprezentuje stan jeszcze nieosiągnięty, zostaje dodany do zbioru węzłów tworzących nową granicę. Węzeł ten zostaje jednak dodany do nowej granicy mimo reprezentowania stanu osiągniętego, jeśli ścieżka łącząca go z korzeniem ma mniejszy koszt niż wszystkie inne ścieżki prowadzące od stanu początkowego (korzenia) do tegoż stanu. Wynikiem działania algorytmu jest węzeł reprezentujący stan docelowy, bądź sygnał o niepowodzeniu (czyli nieznalezieniu ścieżki prowadzącej od stanu początkowego do stanu docelowego). Zarówno szczegóły działania algorytmu, jak i zwracany przezeń wynik, zależne są oczywiście od wyboru funkcji *f* — zajmiemy się tym w dalszej części rozdziału.

LISTING 3.1. Algorytm wyszukiwania według strategii „najpierw najlepszy" i funkcja EXPAND dokonująca rozwijania węzła. Wykorzystywane struktury danych opisujemy w sekcji 3.3.2, znaczenie instrukcji **`yield`** wyjaśnione jest w Dodatku B

```
function BEST-FIRST-SEARCH(problem, f) returns węzeł docelowy albo sygnalizacja niepowodzenia
węzeł ← NODE(STATE=problem.INITIAL)
granica ← kolejka priorytetowa uporządkowana przez funkcję f, początkowo węzeł jest jej jedynym elementem
```

```
osiągnięte ← tabela przeglądowa kojarząca stany z reprezentującymi je węzłami.
                 Początkowo zawiera jeden element — stan problem.INITIAL
                 i reprezentujący go węzeł.

  while not IS-EMPTY(granica) do
    węzeł ← POP(granica)
    if problem.IS-GOAL(węzeł.STATE) then return węzeł
    for each potomny in EXPAND(problem, węzeł) do
      s←potomny.STATE
      if (s nie wchodzi w skład osiągnięte) or potomny.PATH-COST < osiągnięte[s].PATH-COST
      then
        osiągnięte[s]←potomny
        dodaj potomny do granica
  return niepowodzenie

function EXPAND(problem, węzeł) yields węzły
  s ← węzeł.STATE

  for each akcja in problem.ACTIONS(s) do
    s' ← problem.RESULT(s, akcja)
    koszt ← węzeł.PATH-COST + problem.ACTION-COST(s, akcja, s')
    yield NODE(STATE=s', PARENT=węzeł, ACTION=akcja, PATH-COST=koszt)
```

3.3.2. Struktury danych związane z wyszukiwaniem

Algorytmy wyszukiwania wykorzystują struktury danych odzwierciedlających bieżącą postać drzewa wyszukiwawczego. Węzeł tego drzewa reprezentowany jest przez strukturę NODE złożoną z czterech następujących komponentów, oznaczających kolejno:

- STATE — stan reprezentowany przez węzeł;
- PARENT — wskaźnik na węzeł macierzysty, czyli ten, który wygenerował niniejszy węzeł;
- ACTION — akcję, której zastosowanie do węzła macierzystego spowodowało wygenerowanie niniejszego węzła;
- PATH-COST — całkowity koszt ścieżki od stanu początkowego do niniejszego węzła. We wzorach matematycznych odpowiada mu oznaczenie g(*węzeł*).

Poruszanie się wstecz po ścieżce przy użyciu wskaźnika PARENT umożliwia dotarcie do korzenia drzewa i przy okazji prześledzenie historii stanów i akcji poprzedzających bieżący węzeł. Gdy ów bieżący węzeł reprezentuje stan docelowy, taka wycieczka udostępnia kompletną ścieżkę stanowiącą rozwiązanie problemu.

Potrzebujemy także struktury danych reprezentującej granicę drzewa. Doskonale nadają się do tego **kolejki**, zważywszy na następujące operacje wykonywane w ramach algorytmów wyszukiwania:

- IS-EMPTY(*granica*)— zwraca *prawda* wtedy i tylko wtedy, gdy *granica* nie zawiera żadnego węzła;
- POP(*granica*) — usuwa czołowy węzeł *granicy* i zwraca go jako wynik;
- TOP(*granica*) — zwraca jako wynik czołowy węzeł *granicy*, ale w przeciwieństwie do operacji POP nie usuwa go;
- ADD(*węzeł*, *granica*)— dodaje *węzeł* do *granicy*.

W algorytmach wyszukiwania wykorzystywane są trzy rodzaje kolejek:

- **Kolejka priorytetowa** — jej węzłem czołowym jest ten, któremu pewna funkcja ewaluacyjna *f* przypisuje najmniejszą wartość wśród wszystkich węzłów w kolejce. Taka kolejka wykorzystywana jest przez algorytm „najpierw najlepszy".
- **Kolejka FIFO** (ang. *First-In-First-Out* — „pierwszy wchodzi, pierwszy wychodzi") — jej węzłem czołowym jest węzeł dodany najdawniej.
- **Kolejka LIFO** (ang. *Last-In-First-Out* — „ostatni wchodzi, pierwszy wychodzi") — zwana także **stosem** (ang. *stack*); jej węzłem czołowym jest węzeł ostatnio dodany. Jak niebawem pokażemy, ten rodzaj kolejki wykorzystywany jest przez wyszukiwanie w głąb.

Zbiór węzłów reprezentujących stany osiągnięte może być reprezentowany w formie tabeli przeglądowej, której kluczami są węzły, a wartościami odpowiadające tym węzłom stany.

3.3.3. Redundantne ścieżki

Drzewo wyszukiwawcze pokazane w dolnej części rysunku 3.4 obejmuje ścieżkę z Arad do Sibiu i z powrotem do Arad. *Arad* jest więc **stanem powtórzonym** na tej ścieżce, w tym przypadku wskutek wystąpienia **cyklu** (zwanego także **zapętloną ścieżką**). Mimo iż cała przestrzeń liczy tylko 20 stanów, kompletne drzewo wyszukiwawcze jest *nieskończone*, ponieważ nie istnieje ograniczenie co do tego, jak często wolno nam przechodzić przez pętlę.

Cykl jest szczególnym przypadkiem **redundantnej ścieżki**. Z Arad do Sibiu można dostać się dwiema drogami: bezpośrednio (140 mil) lub przez Zerind i Oradea (297 mil). Ta druga ścieżka nie ma jednak znaczenia, jako „gorsza", czyli dłuższa; przejście między tą samą parą stanów — od *Arad* do *Sibiu* — może bowiem dokonać się po krótszej ścieżce. Ścieżka *Arad – Zerind – Oradea – Sibiu* jest ścieżką redundantną — nie należy brać jej pod uwagę przy poszukiwaniu optymalnych ścieżek.

Wyobraźmy sobie agenta poruszającego się po gridzie o rozmiarach 10×10 komórek, z możliwością jego swobodnego (nie ma przeszkód między komórkami) przemieszczania się do każdej z komórek sąsiednich względem bieżącej. Agent znajdujący się na dowolnym polu może dotrzeć do dowolnego innego w co najwyżej 9 ruchach. Ale liczba ścieżek o długości 9 to trochę mniej niż 89 („trochę mniej" ze względu na krawędzie gridu), czyli około 100 milionów. Oznacza to, że do każdej ze 100 komórek gridu prowadzi ok. miliona redundantnych ścieżek o długości 9, ergo — jeśli wyeliminujemy redundantne ścieżki z poszukiwań, możemy skrócić czas wyszukiwania milion razy! Zwykło się mawiać, że *kto zapomina o swej przeszłości, ściąga na siebie klątwę jej powtarzania* i ta uwaga idealnie wpisuje się w algorytmy wyszukiwania. Brzmi groźnie, ale tak naprawdę tylko pozornie, bo złowieszcze fatum można oddalić co najmniej na trzy sposoby.

Po pierwsze, możemy przytoczone ostrzeżenie potraktować dosłownie i zapamiętywać wszystkie osiągane stany (jak ma to miejsce w wyszukiwaniu „najpierw najlepszy"). Pozwala to na wykrywanie wszystkich ścieżek wiodących do poszczególnych stanów i zapamiętywanie dla każdego stanu tylko tej najlepszej. Jest to preferowana metoda dla przestrzeni stanów, w których istnieje wiele redundantnych ścieżek, ale jej ograniczeniem są wymagania pamięciowe związane z zapamiętywaniem osiągniętych stanów — ich tablica może nie mieścić się w pamięci.

Po drugie, możemy wybrać drugą skrajność — oddalić irracjonalną obawę przed „powtarzaniem przeszłości" i potraktować je jako rzecz normalną. Istnieją pewne sformułowania problemów, w których istnienie dwóch ścieżek prowadzących do tego samego stanu jest bardzo rzadkie lub wręcz wykluczone. Rezygnując z wykrywania redundantnych ścieżek, nie musimy zapamiętywać historii osiąganych stanów i zmniejszamy wymagania pamięciowe dla algorytmu. Takim problemem jest na przykład problem montowania jakiegoś mechanizmu czy podzespołu z części (bądź też kompletowanie zamówienia do wysyłki), gdzie każda akcja polega na dodaniu jednej części, ale kolejność dodawania poszczególnych części podlega pewnym ograniczeniom — jeżeli należy dodać pewne dwie części *A* i *B*, to koniecznie musi się to odbyć w kolejności „najpierw *A*, potem *B*", nie odwrotnie, i takich ograniczeń nałożonych na pary części może być bardzo wiele. Algorytm wyszukiwania nazywamy **wyszukiwaniem grafowym**, jeśli prowadzi on kontrolę

redundantnych ścieżek, i **wyszukiwaniem drzewiastym**[7] w przeciwnym razie. Algorytm `BEST-FIRST-SEARCH` z listingu 3.1 to wyszukiwanie grafowe; stanie się on wyszukiwaniem drzewiastym, gdy usuniemy z niego wszystkie odwołania do tablicy *osiągnięte* — algorytm będzie wymagał mniej pamięci, ale jednocześnie będzie działać wolniej.

Po trzecie wreszcie, wykazując pewną dozę ostrożności (licho nie śpi!) możemy pójść na pewien kompromis i ograniczyć się do sprawdzania cykli, rezygnując ze sprawdzania innych ścieżek redundantnych. Cykle możemy łatwo wykrywać bez zaangażowania dodatkowej pamięci, wykorzystując wskaźniki na węzły macierzyste (`PARENT`) w strukturach `NODE` reprezentujących węzły i sprawdzając, czy dany stan nie pojawił się wcześniej w łańcuchu utworzonym przez te wskaźniki. Niektóre implementacje tego pomysłu przeprowadzają kompletną analizę tego łańcucha, wykrywając w ten sposób wszystkie cykle, inne ograniczają się do kilku kroków wstecz (na przykład do rodzica, dziadka, pradziadka, prapradziadka itp.), wykrywając tym samym jedynie cykle, które nie przekraczają założonej długości; zazwyczaj funkcjonują wówczas inne mechanizmy radzące sobie z dłuższymi cyklami.

3.3.4. Wydajność rozwiązywania problemów

Zanim zajmiemy się szczegółami różnych algorytmów wyszukiwania, przyjrzyjmy się podstawowym kryteriom ich oceny — jakościowej i ilościowej — stanowiącym jednocześnie kryteria wyboru konkretnego algorytmu, jako najlepszego w kontekście danego problemu i wymogów dotyczących jego rozwiązania. Najważniejsze spośród tych kryteriów to:

- **Zupełność** (ang. *completeness*) — jeśli istnieje rozwiązanie problemu, algorytm daje gwarancję jego znalezienia; w przypadku braku rozwiązania algorytm daje gwarancję sygnalizacji tego faktu.
- **Optymalność kosztowa** (ang. *cost optimality*) — spośród wszystkich rozwiązań algorytm zwraca to charakteryzujące się najmniejszym kosztem.
- **Złożoność czasowa** (ang. *time-complexity*)[8]— to czas potrzebny na znalezienie rozwiązania, w zależności od rozmiaru problemu: może być wyrażony w jednostkach czasu rzeczywistego lub (bardziej abstrakcyjnie) liczbą analizowanych stanów bądź akcji.
- **Złożoność pamięciowa** (ang. *space-complexity*) — to wielkość pamięci niezbędnej do wykonania algorytmu, w zależności od rozmiaru problemu.

Aby zrozumieć znaczenie zupełności algorytmu, rozważmy problem wyszukiwania z jednym celem. Ten cel może znajdować się w dowolnym miejscu w przestrzeni stanów; dlatego algorytm zupełny musi być w stanie systematycznie badać każdy stan osiągalny ze stanu początkowego. W skończonych przestrzeniach stanów można to osiągnąć dość prosto: badając konsekwentnie wszystkie ścieżki i odrzucając te, które są cyklami (na przykład *Arad – Sibiu – Arad*) ostatecznie dotrzemy do każdego (osiągalnego) stanu.

W nieskończonych przestrzeniach stanów musimy być jednak bardziej ostrożni. Przypomnijmy sobie hipotezę Knutha (ostatni ze standaryzowanych problemów w sekcji 3.2.1). Jedna ze ścieżek, wychodzących ze stanu początkowego „4”, mogłaby prowadzić w nieskończoność przez kolejne złożenia funkcji „silnia” (4, 4!, (4!)!, ((4!)!) itd.). Analogicznie mają się sprawy w przypadku agenta poruszającego się po *nieskończonym* gridzie (bez przeszkód między komórkami) — jedna z (nieskończonych) ścieżek reprezentuje ruch po linii prostej. W obu przypadkach algorytm nie powraca do stanów już osiągniętych, ale jego eksplorowanie przestrzeni stanów nigdy się nie skończy. Nie jest to więc algorytm zupełny.

---

[7] Sformułowanie „wyszukiwanie drzewiaste” bierze się stąd, że choć przestrzeń stanów reprezentowana jest przez graf, to w procesie wyszukiwania traktujemy go tak, *jakby był drzewem* — czyli jakby dla każdego stanu istniała dokładnie jedna ścieżka łącząca go ze stanem początkowym.

[8] Synonimiczne określenia to „dopuszczalność” (ang. *admissibility*) i (po prostu) „optymalność” — to ostanie jest jednak mylące, bo może odnosić się do optymalności według innego kryterium.

Aby algorytm wyszukiwania był kompletny, musi być **systematyczny** w tym sensie, że eksplorując nieskończoną przestrzeń stanów, osiągnie ostatecznie dowolny stan połączony ze stanem początkowym. Na przykład dla nieskończonego gridu systematyczna eksploracja komórek, rozpoczynająca się od pewnej komórki stanowiącej stan początkowy. Mogłaby przebiegać wzdłuż spiralnej linii przebiegającej w kierunku zegarowym najpierw komórki sąsiednie wspomnianej komórki początkowej, potem komórki odległe o 2 pozycje od tejże, potem o 3 itd. Niestety, jeżeli *żadna* komórka gridu nie będzie rozwiązaniem problemu, algorytm będzie kreślił nieskończoną spiralę i nigdy nie zwróci informacji o braku rozwiązania.

Złożoność czasową i złożoność pamięciową rozpatruje się w odniesieniu do rozmiaru problemu, czyli pewnego aspektu jego trudności. W informatyce teoretycznej za rozmiar problemu wyszukiwania przyjmuje się zwykle wielkość grafu przestrzeni stanów, czyli sumę wierzchołków i krawędzi tego grafu ($|V|+|E|$) — czyli sumaryczną liczbę stanów i par (stan, akcja możliwa w tym stanie). Jest to odpowiednie w sytuacji, gdy wspomniany graf ma postać jawnie określonej struktury, takiej jak mapa Rumunii z rysunku 3.1. W wielu problemach sztucznej inteligencji graf przestrzeni stanów określony jest *niejawnie*, poprzez stan początkowy, akcje i model przejścia. W takim przypadku złożoność problemu można mierzyć w kategoriach **głębokości** (ang. *depth*) *d,* czyli liczby akcji w optymalnym rozwiązaniu, maksymalnej liczby akcji na dowolnej ścieżce *m* lub **czynnika rozgałęzienia** (ang. *branching factor*) *b*, czyli liczby następników węzła wymagającego analizy.

3.4. Strategie wyszukiwania niedoinformowanego

Algorytm wyszukiwania, w którym niemożliwe jest określenie, jak bardzo odległy jest dany stan od celu (celów), nazywamy algorytmem **niedoinformowanym** (ang. *uninformed*). Nasz agent znajdujący się w Arad, zamierzający dotrzeć do Bukaresztu, nieznający geografii Rumunii, nie wie, które z sąsiednich miast — Zerind czy Sibiu — znajduje się bliżej celu jego podróży i do którego z nich się udać. Z kolei agent poinformowany (patrz podrozdział 3.5) wiedziałby, że znacznie bliżej Bukaresztu znajduje się Sibiu, a więc prawdopodobnie prowadząca przezeń trasa będzie krótsza niż ta wiodąca przez Zerind.

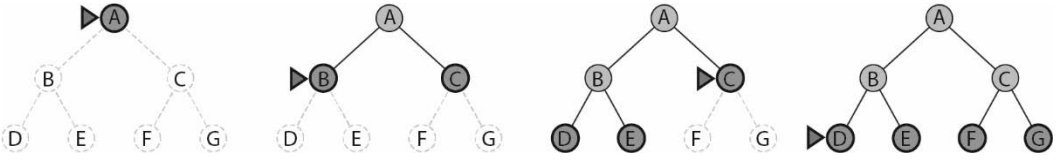
3.4.1. Wyszukiwanie wszerz

Gdy wszystkie akcje mają taki sam koszt, odpowiednią strategią wyszukiwania jest **wyszukiwanie wszerz** (ang. *breadth-first search*), zgodnie z którym bezpośrednio po rozwinięciu danego węzła rozwijane są wygenerowane właśnie węzły potomne — zasadę tę stosuje się rekurencyjnie, poczynając od korzenia. Algorytm stosujący tę strategię jest systematyczny i jest zupełny nawet w nieskończonych przestrzeniach stanów. Algorytm wyszukiwania wszerz możemy uzyskać z algorytmu `BEST-FIRST-SEARCH` z listingu 3.1 poprzez zdefiniowanie funkcji ewaluacyjnej $f(n)$ jako głębokości węzła n, czyli akcji potrzebnych do jego osiągnięcia.

Jednak dzięki kilku sztuczkom możemy uzyskać algorytm znacznie bardziej efektywny — bardziej efektywną od kolejki priorytetowej okazuje się bowiem w tej sytuacji kolejka FIFO, zapewniająca właściwą kolejność przetwarzania węzłów. Nowe węzły (które oczywiście mają większą głębokość niż ich węzeł macierzysty) wstawiane są na koniec kolejki, dzięki czemu elementy tej kolejki uporządkowane są rosnąco względem głębokości — węzły są więc przetwarzane (rozwijane) począwszy od „najpłytszych". Co więcej, struktura *osiągnięte*, w algorytmie `BEST-FIRST-SEARCH` będąca tabelą przeglądową mapującą stany na węzły, tym razem jest jedynie zbiorem stanów; jest to wystarczające, ponieważ gdy osiągniemy dany stan, nigdy później nie znajdziemy już krótszej ścieżki prowadzącej do niego[9]. Kapitalną tego konsekwencją jest możliwość **wczesnej weryfikacji celu** — jeżeli stan reprezentowany przez nowo wygenerowany

[9] Czyli ścieżki o mniejszym koszcie, jako że wszystkie akcje mają taki sam koszt — *przyp. tłum.*

węzeł okaże się stanem docelowym, będzie takim stanem osiągniętym po możliwie najkrótszej (najtańszej) ścieżce, *czyli będzie rozwiązaniem problemu*. W algorytmie BEST-FIRST-SEARCH taka weryfikacja możliwa jest dopiero po pobraniu węzła z kolejki, jest więc przykładem **późnej weryfikacji celu**. Rysunek 3.7 przedstawia przebieg wyszukiwania wszerz w drzewie binarnym, zaś na listingu 3.2 widoczny jest kod algorytmu BREADRH-FIRST-SEARCH — zwróć uwagę na podobieństwa i różnice w stosunku do BEST-FIRST-SEARCH.



RYSUNEK 3.7. Wyszukiwanie wszerz w prostym drzewie binarnym. Na każdym etapie węzeł przeznaczony do rozwinięcia zaznaczony jest trójkątnym wskaźnikiem

LISTING 3.2. Algorytmy wyszukiwania wszerz i wyszukiwania przy jednolitym koszcie

```
function BREADTH-FIRST-SEARCH(problem) returns węzeł docelowy albo sygnalizacja niepowodzenia
  węzeł ← NODE(problem.INITIAL)
  if problem.IS-GOAL(węzeł.STATE) then return węzeł
  granica ← kolejka FIFO, początkowo węzeł jest jej jedynym elementem
  osiągnięte ←{problem.INITIAL}
  while not IS-EMPTY(granica) do
    węzeł ← POP(granica)
    for each potomny in EXPAND(problem, węzeł) do
      s←potomny.STATE
      if problem.IS-GOAL(s) then return potomny
      if (s nie wchodzi w skład osiągnięte) then
          dodaj s do osiągnięte
          dodaj potomny do granica
  return niepowodzenie

function UNIFORM-COST-SEARCH(problem) returns węzeł docelowy albo sygnalizacja niepowodzenia
  return BEST-FIRST-SEARCH(problem, PATH-COST)
```

Wyszukiwanie wszerz zawsze znajduje rozwiązanie z minimalną liczbą akcji, ponieważ gdy generowany jest węzeł na głębokości d, wygenerowane są już wszystkie węzły na głębokości d–1; gdyby któryś z tych ostatnich był rozwiązaniem, algorytm zauważyłby ten fakt i zwrócił to rozwiązanie. Jeśli wszystkie akcje na ścieżce będącej rozwiązaniem mają jednakowy koszt, to ścieżka ta (jako przecież najkrótsza) jest jednocześnie ścieżką o najmniejszym koszcie, zatem *w takiej sytuacji algorytm wyszukiwania wszerz jest algorytmem optymalnym kosztowo*. Niezależnie od jednolitego czy zróżnicowanego kosztu akcji, *algorytm wyszukiwania wszerz jest algorytmem zupełnym*. Aby oszacować jego złożoność (czasową i pamięciową), wyobraźmy sobie wyszukiwanie drzewa, którego każdy węzeł (który nie jest liściem) ma dokładnie b węzłów potomnych. Korzeń drzewa generuje b węzłów, z których każdy generuje b kolejnych węzłów itd.; gdy wygenerowane zostaną wszystkie węzły na głębokości d, ogólna liczba wygenerowanych węzłów wynosi

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

Wszystkie węzły egzystują jednocześnie w pamięci, zatem zarówno czasowa, jak i pamięciowa złożoność algorytmu jest rzędu $O(b^d)$. Konkretne wartości kryjące się za *wykładniczą* złożonością mogą być porażające. Wyobraźmy sobie typowy problem ze świata rzeczywistego, w którym czynnik rozgałęzienia wynosi $b = 10$; problem ten rozwiązywany jest za pomocą komputera, którego procesor zdolny jest generować milion węzłów w ciągu sekundy, a każdy węzeł wymaga 1 kilobajta pamięci. Dojście algorytmu do głębokości $d = 10$ zajmie procesorowi niecałe 3 godziny, lecz wymagania pamięciowe sięgną wówczas 10 terabajtów[10] — zatem *nie czas wyszukiwania, ale wymagania pamięciowe są prawdziwym problemem w przypadku algorytmu wyszukiwania wszerz*. Złożoności czasowej nie należy bynajmniej lekceważyć — przy głębokości $d = 14$ wyszukiwanie (nawet przy zaspokojeniu wymagań pamięciowych rzędu 100 000 terabajtów) zajęłoby 3,5 roku. Jak widać, *problemy wyszukiwania o złożoności wykładniczej nie dadzą się rozwiązywać za pomocą algorytmów niedoinformowanych, z wyjątkiem być może bardzo prostych przypadków.*

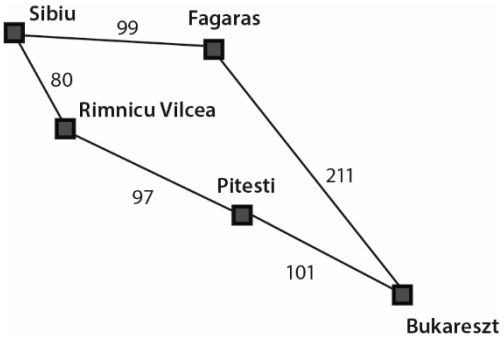
3.4.2. Algorytm Dijkstry — wyszukiwanie przy jednolitym koszcie

Gdy koszty akcji są zróżnicowane, oczywistym wyborem wydaje się użycie wyszukiwania „najpierw najlepszy" z funkcją ewaluacyjną $f(n)$ zwracającą koszt ścieżki rozpoczynającej się w korzeniu, a kończącej w węźle n. Wynalazcą tego algorytmu jest Edsger Wybe Dijkstra (1930 – 2002), holenderski pionier informatyki; oczywiście z tego względu społeczność informatyków nazywa ten algorytm „algorytmem Dijkstry" (choć to tylko jeden z wielu cennych wkładów Dijkstry w rozwój informatyki). Na gruncie sztucznej inteligencji algorytm ten nosi nazwę **wyszukiwania przy jednolitym koszcie** (ang. *uniform-cost search*). Podczas gdy algorytm wyszukiwania wszerz porusza się po drzewie wyszukiwawczym „falą" przebiegającą kolejne głębokości — $d = 1$, $d = 2$ itd. — algorytm Dijkstry rozprzestrzenia się po drzewie poprzez ścieżki o jednolitym koszcie. Funkcję `UNIFORM-COST-SEARCH` realizująca ten algorytm otrzymamy, wywołując funkcję `BEST-FIRST-SEARCH` z listingu 3.1 z funkcją ewaluacyjną, która dla węzła będącego jej argumentem zwraca wartość jego atrybutu `PATH-COST` — co pokazaliśmy w dolnej części listingu 3.2.

Spójrzmy na rysunek 3.8., ilustrujący problem dostania się z Sibiu do Bukaresztu. Węzeł Sibiu ma dwa następniki: Rimnicu Vilcea i Fagaras o kosztach (odpowiednio) 80 i 99. Wybieramy węzeł o mniejszym koszcie (Rimnicu Vilcea) i rozwijamy go, dochodząc do węzła Pitesti (nie mamy innego wyboru), którego koszt wynosi 80+97 = 177. Węzeł Fagaras ma mniejszy koszt (99), więc rozwijamy go, dochodząc do Bukaresztu, czyli węzła o koszcie 99+211 = 310. Węzeł ten zostaje dodany do *granicy*; choć reprezentuje on stan docelowy, algorytm tego nie zauważa — sprawdzanie, czy węzeł reprezentuje stan docelowy, następuje *dopiero po pobraniu tego węzła z granicy za pomocą operacji POP*, nie w momencie dodania go do *granicy*.

Algorytm kontynuuje pracę, przechodząc do węzła Pitesti i rozwijając go, co daje nową ścieżkę do Bukaresztu, o koszcie 80+97+101 = 278. Jest to koszt mniejszy od poprzedniego (310), więc nowa ścieżka zastępuje poprzednią w tablicy *osiągnięte*, kończący ją węzeł Bukareszt zostaje dodany do *granicy*. Jako węzeł o najmniejszym ze wszystkich koszcie zostanie wkrótce pobrany z granicy (za pomocą operacji POP) i rozpoznany jako rozwiązanie. Zwróćmy uwagę na ważny fakt: gdybyśmy testowali węzeł na okoliczność reprezentowania stanu docelowego już w momencie dodawania tego węzła do *granicy*, algorytm zwróciłby ścieżkę wiodącą przez Rimnicu Vilcea i Pitesti, która nie jest najkrótsza.

[10] Dzięki pamięci wirtualnej program może uzyskać tak dużą przestrzeń adresową, nawet jeśli fizyczna pamięć RAM komputera jest znacznie mniejsza, na przykład liczona w gigabajtach. Zwykle oznacza to jednak drastyczną degradację wydajności, spowodowaną intensywną wymianą danych między pamięcią RAM a plikiem wymiany — być może nawet do poziomu *jednego* węzła na sekundę (zamiast wspomnianego miliona). A wtedy szacowane 3 godziny rozciągną się na *kilka tygodni*. Nota bene to kolejne świadectwo wymagań, jakie badania nad sztuczną inteligencją stawiają przed infrastrukturą sprzętową i programową — *przyp. tłum.*



RYSUNEK 3.8. Część przestrzeni stanów podróży po Rumunii, wybrana dla zilustrowania wyszukiwania przy jednolitym koszcie

Złożoność algorytmu wyszukiwania przy jednolitym koszcie rozpatruje się w kategoriach C^* — kosztu optymalnego rozwiązania[11] — oraz ε — dolnego ograniczenia kosztu każdej akcji, $\varepsilon > 0$. Złożoność — czasowa i pamięciowa — najgorszego przypadku wynosi wówczas $O\left(b^{1+\lfloor C^*/\varepsilon \rfloor}\right)$, czyli może być znacznie większa od b^d ($\lfloor x \rfloor$ oznacza największą liczbę całkowitą nieprzekraczająca x). Jest tak dlatego, że algorytm wyszukiwania przy jednolitym koszcie może eksplorować duże drzewa „tanich" akcji przed zbadaniem kosztownych ścieżek zawierających być może użyteczne akcje. Gdy koszty wszystkich akcji są jednakowe, wartość $b^{1+\lfloor C^*/\varepsilon \rfloor}$ równa jest b^{d+1} i wyszukiwanie przy jednolitym koszcie równoważne jest wyszukiwaniu wszerz.

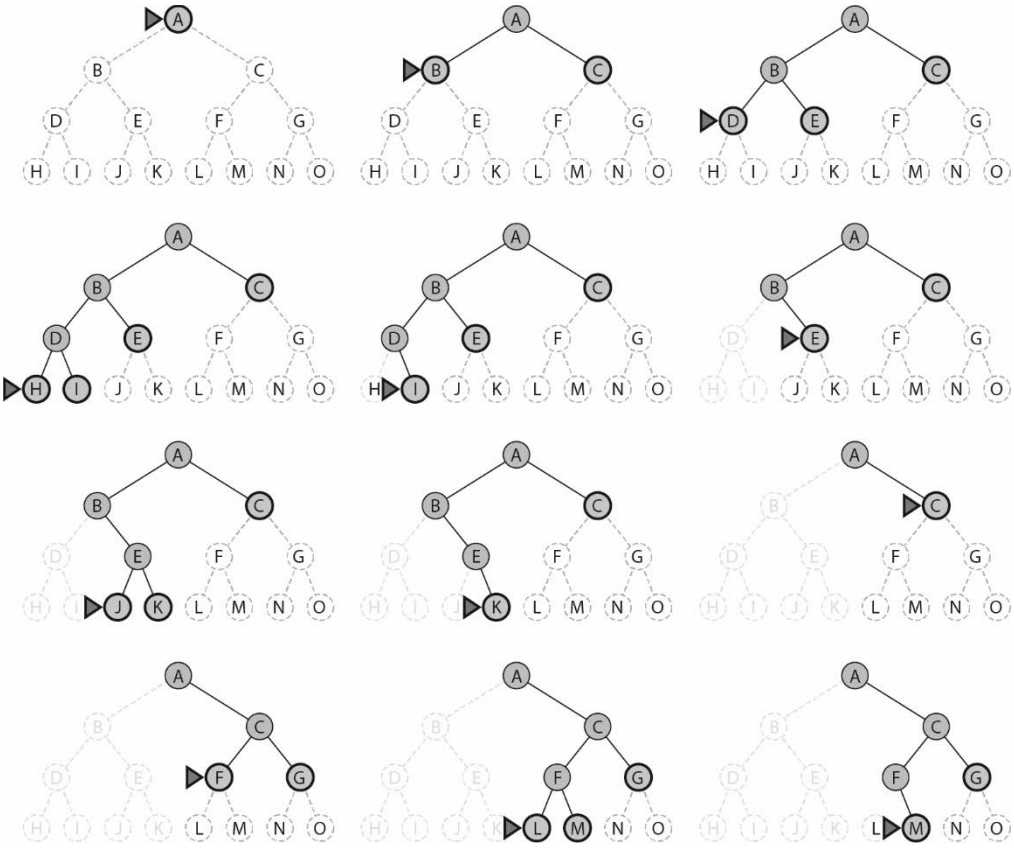
Algorytm wyszukiwania przy jednolitym koszcie jest algorytmem zupełnym i optymalnym kosztowo, ponieważ pierwsze znalezione przez niego rozwiązanie ma koszt nie większy niż koszt każdego innego węzła należącego do granicy. Algorytm ten systematycznie przegląda wszystkie ścieżki w kolejności wzrastających kosztów, nigdy nie dając się złapać w pojedynczą nieskończoną ścieżkę, pod warunkiem, że koszt każdej akcji jest większy od pewnego dodatniego ε.

3.4.3. Wyszukiwanie w głąb i problemy z pamięcią

Algorytm **wyszukiwania w głąb** (ang. *depth-first search*) zawsze rozwija w pierwszej kolejności *najgłębszy* węzeł na granicy. Można go otrzymać przez wywołanie algorytmu `BEST-FIRST-SEARCH` z funkcją ewaluacyjną $f(n)$ zwracającą *zanegowaną głębokość* węzła n, zwykle jednak implementuje się go jako wyszukiwanie drzewiaste, nie grafowe, nieutrzymujące tablicy osiągniętych stanów. Postęp przykładowego wyszukiwania ilustruje rysunek 3.9; po rozwinięciu węzła wyszukiwanie przechodzi natychmiast do jego węzłów potomnych, po czym „cofa się" do najbliższego najgłębszego węzła, który nadal posiada nierozwinięte węzły potomne. Algorytm wyszukiwania w głąb *nie jest optymalny kosztowo* — zwraca pierwsze znalezione rozwiązanie nawet jeśli nie jest najtańsze.

W skończonych przestrzeniach stanów o strukturze drzewiastej algorytm jest wydajny i zupełny — w przestrzeni acyklicznej, nawet jeśli dany stan będzie rozwijany wielokrotnie (w wyniku przemierzania różnych ścieżek prowadzących do niego), ale ostatecznie i tak odwiedzony zostanie każdy stan.

[11] W tym miejscu i w dalszym ciągu książki gwiazdka towarzysząca wielkości C oznacza optymalną wartość tej wielkości.



RYSUNEK 3.9. Dwanaście kroków (od lewej do prawej, z góry na dół) przebiegu wyszukiwania w głąb w drzewie binarnym, począwszy od stanu *A*, skończywszy na stanie docelowym *M*. Kolorem zielonym oznaczono węzły tworzące bieżącą granicę, trójkątny wskaźnik identyfikuje węzeł do aktualnego rozwijania. Węzły już rozwinięte oznaczone są kolorem fioletowym, węzły stanowiące kandydatury do następnych rozwinięć są niepokolorowane. Słabiej widoczne rozwinięte węzły niemające węzłów potomnych w granicy wykluczane są z dalszych poszukiwań

W przestrzeni stanów zawierającej cykle algorytm może utknąć w nieskończonej pętli, dlatego niektóre jego implementacje sprawdzają każdy nowy węzeł pod kątem cykli. Wreszcie — w nieskończonej przestrzeni stanów algorytm może utknąć na nieskończonej ścieżce, nawet jeśli przestrzeń ta nie zawiera cykli, nie jest więc zupełny w nieskończonych przestrzeniach.

Dlaczego więc w ogóle sięgać po wyszukiwanie w głąb, skoro dostępne są wyszukiwanie wszerz i wyszukiwanie „najpierw najlepszy”? Otóż w przypadku problemów, do których można zastosować wyszukiwanie drzewiaste, wyszukiwanie w głąb cechuje się znacznie mniejszym zapotrzebowaniem na pamięć niż jego wspomniani konkurenci: nie ma potrzeby utrzymywania tablicy stanów osiągniętych, także granice są zwykle mniej liczebne. Jeżeli mianowicie wyobrazimy sobie granice wyszukiwania wszerz jako rozszerzającą się sferę, to granica wyszukiwania w głąb będzie w tej analogii wydłużającym się promieniem tej sfery.

W przypadku skończonej przestrzeni stanów o strukturze drzewa (takiego jak na rysunku 3.9) wyszukiwanie w głąb zajmuje czas proporcjonalny do liczby stanów, a jego wymagania pamięciowe są rzędu $O(bm)$, gdzie b jest czynnikiem rozgałęzienia, a m największą głębokością węzła w drzewie. Dzięki temu niektóre problemy, które przy wyszukiwaniu wszerz wymagałyby eksabajtów pamięci, można za pomocą wyszukiwania w głąb rozwiązywać kosztem zaledwie kilobajtów. Właśnie ze względu na ową oszczędność pamięci drzewiaste wyszukiwanie w głąb zostało powszechnie przyjęte w wielu obszarach sztucznej inteligencji jako przysłowiowy wół roboczy, między innymi w zadaniach dotyczących spełniania ograniczeń (patrz rozdział 6.), spełnialności w rachunku zdań (rozdział 7.) i programowania w logice (rozdział 9.).

Jeszcze skromniejszymi wymogami pamięciowymi charakteryzuje się odmiana wyszukiwania w głąb, zwana **wyszukiwaniem z nawrotami** (ang. *backtracking search*), którą szczegółowo opisujemy w rozdziale 6. W procesie rozwijania węzła generowany jest każdorazowo tylko jeden węzeł potomny (zamiast wszystkich), w węźle częściowo rozwiniętym przechowywana jest informacja o tym, który z węzłów potomnych wygenerować jako następny. Ponadto generowanie kolejnych węzłów potomnych odbywa się poprzez *modyfikowanie* opisu stanu w węźle macierzystym, zamiast przydzielania pamięci do zapamiętywania nowych stanów. Konieczność zapamiętywania tylko jednego stanu dla każdego węzła redukuje wymagania pamięciowe z $O(bm)$ do $O(m)$. Możliwe jest także kontrolowanie stanu bieżącej ścieżki za pomocą efektywnej struktury (w postaci zbioru), dzięki czemu wykrywanie cyklu następuje w czasie $O(1)$ (czyli natychmiast) zamiast w czasie $O(m)$. Warunkiem koniecznym do zastosowania wyszukiwania z nawrotami jest możliwość *anulowania* wykonanych już akcji (to właśnie stanowi istotę nawrotów). Wyszukiwanie z nawrotami ma krytyczne znaczenie dla pomyślnego rozwiązywania problemów charakteryzujących się rozbudowanym opisem stanu, na przykład zrobotyzowanych linii montażowych.

3.4.4. Wyszukiwanie z ograniczeniem głębokości i iteracyjne zagłębianie

Aby wykluczyć ryzyko utknięcia wyszukiwania w głąb na nieskończonej ścieżce, można **ograniczyć głębokość wyszukiwania**. Przyjmujemy mianowicie pewien limit głębokości λ i wszystkie węzły na głębokości λ traktujemy tak, jakby nie miały węzłów potomnych (patrz listing 3.3). Złożoność czasowa takiego wyszukiwania wynosi $O(b^{\lambda})$, a złożoność pamięciowa $O(b\,\lambda)$. Niestety, jeżeli niefortunnie wybierzemy λ, algorytm może nie znaleźć rozwiązania (mimo iż ono istnieje), zatem stanie się niezupełny.

Ponieważ wyszukiwanie w głąb jest wyszukiwaniem drzewiastym, nie możemy generalnie zapobiec marnowaniu czasu na analizę redundantnych ścieżek, ale możemy eliminować cykle, za cenę dodatkowego czasu obliczeń. Możemy wykrywać większość cykli, posuwając się jedynie o kilka pozycji w łańcuchu węzłów macierzystych, dłuższe cykle eliminowane będą automatycznie ze względu na ograniczenie głębokości wyszukiwania.

W sensownym wyborze ograniczenia głębokości (λ) może dopomóc wstępna analiza problemu. Skoro na przykład na mapie z rysunku 3.1 znajduje się 20 miast, to rozsądnym wyborem okazuje się $\lambda = 19$. Gdybyśmy jednak bliżej przyjrzeli się problemowi zauważylibyśmy, że z każdego miast można dotrzeć do dowolnego innego w maksymalnie 9 etapach. Liczba ta, nazywana **średnicą grafu** przestrzeni stanów, daje nam lepsze ograniczenie, skutkujące bardziej efektywnym wyszukiwaniem. Dla większości problemów właściwe ograniczenie staje się znane dopiero… po rozwiązaniu problemu.

Wyszukiwanie z iteracyjnym zagłębianiem (ang. *iterative deepening search*) to strategia doboru właściwej wartości λ metodą prób i błędów — $\lambda = 0$, $\lambda = 1$ itd. aż do skutku, czyli znalezienia rozwiązania albo zwrócenia przez algorytm sygnału *niepowodzenia* (zamiast sygnału przekroczenia *limitu* głębokości). Algorytm takiego zagłębiania widoczny jest na listingu 3.3. Iteracyjne zagłębianie wyszukiwania łączy w sobie zalety wyszukiwania w głąb i wyszukiwania wszerz. Podobnie jak wyszukiwanie w głąb, cechuje się skromnymi wymaganiami pamięciowymi — $O(bd)$ gdy istnieje rozwiązanie i $O(bm)$ przy jego braku (m jest liczbą stanów w przestrzeni). Podobnie jak wyszukiwanie wszerz, jest optymalne kosztowo dla problemów, w których wszystkie akcje mają taki sam koszt; jest ponadto zupełne w skończonych przestrzeniach stanów — acyklicznych oraz w przypadku, gdy analizujemy węzły pod kątem cykli, poruszając się w górę w łańcuchu węzłów macierzystych.

LISTING 3.3. Iteracyjne zagłębianie wyszukiwania (ITERATIVE-DEEPENING-SEARCH) zaimplementowane jako powtarzane wywoływanie wyszukiwania z ograniczeniem głębokości (DEPTH-LIMITED-SEARCH) dla coraz większych limitów głębokości. Wynik zwracany przez tę ostatnią technikę może być trojaki: węzeł reprezentujący stan docelowy, sygnalizacja przekroczenia limitu głębokości albo sygnalizacja braku rozwiązania po eksploracji wszystkich węzłów. Zwróćmy uwagę na brak struktury przechowującej stany osiągnięte, co znacznie redukuje zapotrzebowanie na pamięć (w porównaniu z wyszukiwaniem „najpierw najlepszy"), ale jednocześnie niesie ryzyko wielokrotnego eksplorowania tych samych stanów na różnych ścieżkach. Ponadto, jeśli funkcja IS-CYCLE nie wykrywa wszystkich cykli, algorytm może utknąć w nieskończonej pętli

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns rozwiązanie albo sygnalizacja niepowodzenia
  for głębokość = 0 to ¥ do
    wynik ← DEPTH-LIMITED-SEARCH(problem, głębokość)
    if wynik ≠ przekroczenie_limitu then return wynik

function DEPTH-LIMITED-SEARCH(problem, λ ) returns węzeł wynikowy albo sygnalizacja niepowodzenia
                                                  albo sygnalizacja przekroczenia limitu głębokości
  granica ← kolejka LIFO (stos) początkowo z węzłem NODE(problem.INITIAL) jako jedynym elementem
  wynik ← niepowodzenie

  while not IS-EMPTY(granica) do
    węzeł ← POP(granica)
    if problem.IS-GOAL(węzeł.STATE) then return węzeł
    if DEPTH(węzeł) > λ then
      wynik ←przekroczenie_limitu
    else if not IS-CYCLE(węzeł) then
      for each potomny in EXPAND(problem, węzeł) do
        dodaj potomny do granica
  return wynik
```

Złożoność czasowa wyszukiwania z iteracyjnym zagłębianiem wynosi $O(b^d)$, gdy istnieje rozwiązanie, i $O(b^m)$, gdy nie ma żadnego. W każdej iteracji generowany jest nowy poziom węzłów, podobnie jak wyszukiwanie wszerz, ale jest istotna różnica: wyszukiwanie wszerz przechowuje w pamięci wszystkie węzły, podczas gdy iteracyjne zagłębianie ponownie przechodzi przez wszystkie poziomy, oszczędzając w ten sposób pamięć kosztem czasu obliczeń. Na rysunku 3.10 widoczne są cztery iteracje zagłębiania w drzewie binarnym, rozwiązanie znalezione zostaje w czwartej iteracji.

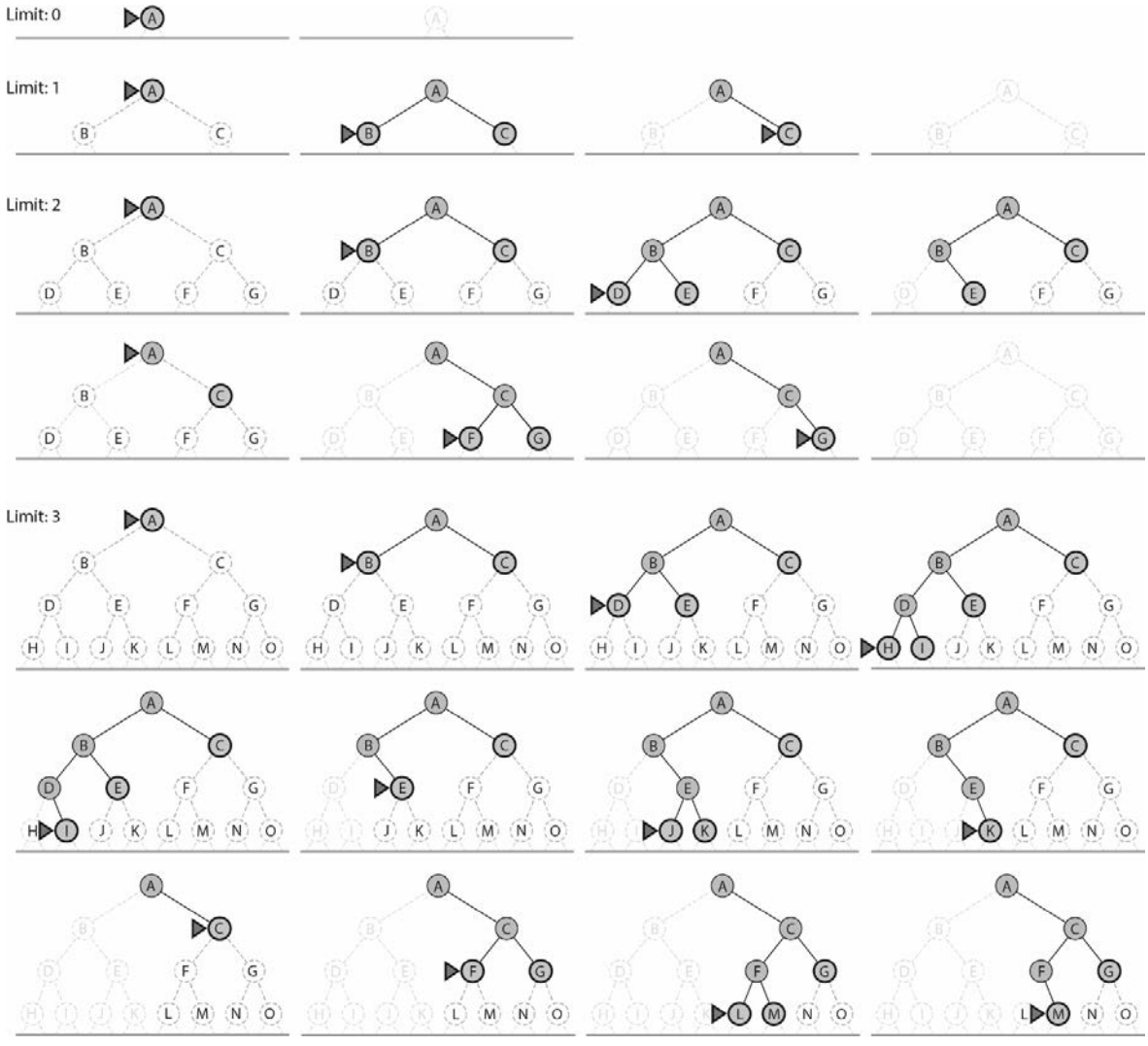
Wielokrotne odwiedzanie początkowych węzłów drzewa w ramach iteracyjnego zagłębiania może wydawać się marnotrawieniem czasu; przestrzenie stanów wielu problemów mają jednak tę cechę, że znakomita większość węzłów w ich drzewach wyszukiwawczych znajduje się na głębszych poziomach i czas stracony na ponowną eksplorację płytszych węzłów nie jest aż tak znaczący. Węzły najniższego poziomu (d) generowane są tylko raz, węzły poziomu bezpośrednio wyższego — dwukrotnie itd., węzły potomne korzenia generowane są d razy. Całkowita liczba aktów generowania węzłów wynosi więc w najgorszym przypadku

$$N(\text{IDS}) = (d)b^1+(d-1)b^2+(d-2)b^3+\ldots+b^d$$

co jest złożonością rzędu $O(b^d)$, asymptotycznie równą tej z wyszukiwania „najpierw najlepszy", na przykład dla $b = 10$, $d = 5$ mamy[12]

$$N(\text{IDS}) = 50+400+3\,000+20\,000+100\,000 = 123\,450$$
$$N(\text{BFS}) = 10+100+1\,000+10\,000+100\,000 = 111\,110$$

[12] IDS to skrót od *Iterative Deepening Search* (przeszukiwanie z zagłębianiem), BFS to skrót od *Best-First Search* (wyszukiwanie „najpierw najlepszy") — *przyp. tłum.*



RYSUNEK 3.10. Cztery iteracje wyszukiwania z zagłębianiem w drzewie binarnym, z limitem głębokości zmieniającym się od 0 do 3, stan docelowy reprezentowany jest przez węzeł *M*. Zauważmy, że węzły wewnętrzne tworzą pojedynczą ścieżkę. Trójkąty wskazują węzły do rozwinięcia; zielone węzły z ciemną obwódką tworzą granicę, węzły słabiej widoczne wykluczone są jako rozwiązania na danym poziomie zagłębienia

Jeśli jednak strata czasu na wielokrotne generowanie tych samych węzłów jest prawdziwym problemem, można użyć następującej strategii hybrydowej: uruchamiamy wyszukiwanie wszerz i kontynuujemy aż do wyczerpania dostępnej pamięci, a następnie przełączamy się na iteracyjne zagłębianie wszystkich węzłów na granicy. *Ogólnie rzecz biorąc, wyszukiwanie z zagłębianiem jest preferowaną metodą wyszukiwania niedoinformowanego w sytuacji, gdy liczebność przestrzeni stanów przekracza możliwości pamięciowe, a głębokość, na jakiej znajduje się węzeł-rozwiązanie, nie jest znana.*

3.4.5. Wyszukiwanie dwukierunkowe

W dotychczas omawianych algorytmach wyszukiwanie rozpoczyna się w stanie początkowym i kończy w jednym ze stanów docelowych. Podejście alternatywne, zwane **wyszukiwaniem dwukierunkowym** (ang. *bidirectional search*), polega na uruchomieniu równoległych wyszukiwań: jednego rozpoczynającego się od stanu początkowego, podążającego w przód, i drugiego rozpoczynającego się od węzła docelowego, podążającego wstecz — w nadziei, że te dwa wyszukiwania spotkają się gdzieś pośrodku drzewa. Motywacja takiego podejścia jest oczywista: złożoność dwukierunkowego wyszukiwania równa $b^{d/2} + b^{d/2} = O(b^{d/2})$ jest znacząco niższa niż $O(b^d)$ w wyszukiwaniu jednokierunkowym, co na przykład dla $b = d = 10$ oznacza skrócenie czasu wykonywania około 50 000 razy.

Aby zaimplementować tę strategię, musimy śledzić dwie granice i utrzymywać dwie tablice stanów osiągniętych, a ponadto zapewnić sobie możliwość „rozumowania wstecz" — jeśli wiemy, że stan *s'* jest następnikiem stanu *s* przy wyszukiwaniu w przód, to jednocześnie musimy wiedzieć, że stan *s* jest następnikiem stanu *s'* przy wyszukiwaniu wstecz. Rozwiązanie otrzymamy, gdy zderzą się obie granice[13]

Istnieje wiele różnych wersji wyszukiwania dwukierunkowego, tak samo jak istnieje wiele różnych algorytmów wyszukiwania jednokierunkowego. W tej sekcji opisujemy dwukierunkowy wariant wyszukiwania „najpierw najlepszy". Chociaż istnieją dwie oddzielne granice, węzeł, który ma zostać rozwinięty jako następny, jest zawsze węzłem z minimalną wartością funkcji ewaluacyjnej, na którejkolwiek z tych granic. Gdy wartością funkcji ewaluacyjnej jest koszt ścieżki, otrzymujemy dwukierunkowy wariant wyszukiwania przy jednolitym koszcie, a jeśli koszt optymalnej ścieżki wynosi C^*, nie będzie rozwijany żaden węzeł o koszcie większym niż $\frac{C^*}{2}$. Możemy dzięki temu uzyskać znaczne przyspieszenie.

Ogólny algorytm wyszukiwania „najpierw najlepszy" w wersji dwukierunkowej widoczny jest na listingu 3.4. Przekazujemy dwie wersje problemu i funkcji ewaluacyjnej, dla wyszukiwania w przód i wyszukiwania wstecz, identyfikowane przyrostkami (odpowiednio) *_F* i *_B*. Kiedy wynikiem funkcji ewaluacyjnej jest koszt ścieżki, wiemy, że pierwsze znalezione rozwiązanie będzie rozwiązaniem optymalnym, ale nie musi to być prawda w sytuacji, gdy mamy dwie różne funkcje ewaluacyjne. Musimy zatem śledzić najlepsze rozwiązanie znalezione do tej pory i wielokrotnie je aktualizować, dopóki funkcja `TERMINATED` nie zwróci wartości *prawda* oznaczającej, że nie istnieje już lepsze rozwiązanie.

LISTING 3.4. Dwukierunkowa odmiana wyszukiwania „najpierw najlepszy" wykorzystuje dwie kolejki priorytetowe implementujące granice i dwie tabele przeglądowe rejestrujące stany osiągnięte i reprezentujące je węzły. Gdy ścieżka reprezentowana przez jedną ze wspomnianych granic osiąga stan, który został już osiągnięty na ścieżce reprezentowanej przez drugą granicę, obie ścieżki zostają połączone (za pomocą funkcji JOIN-NODES) w jedno rozwiązanie. Pierwsze otrzymane rozwiązanie niekoniecznie jest rozwiązaniem optymalnym, więc kontynuowane jest poszukiwanie następnych aż do momentu, gdy funkcja TERMINATED zasygnalizuje brak takowych, zwracając wartość prawda

```
function BIBF-SEARCH(problem_F, f_F, problem_B, f_B) returns węzeł będący rozwiązaniem
                                                          albo sygnalizacja niepowodzenia
    // węzeł reprezentujący stan początkowy
  węzeł_F ← NODE(problem_F.INITIAL)
```

---

[13] Przeszukiwanie dwukierunkowe kończy się, gdy spotykają się granice dla obu kierunków (*granica_F* i *granica_B*), czyli gdy okazuje się, że mają one wspólny węzeł. Tak się jednak składa, że kolejka priorytetowa (bo w takiej postaci implementowane są obie granice) nie udostępnia funkcji sprawdzania, czy dany węzeł jest jej elementem; funkcję taką oferuje natomiast tablica przeglądowa rejestrująca stany osiągnięte. Zatem, choć koncepcyjnie testujemy, czy dwie granice mają niepustą część wspólną, w praktyce zmuszeni jesteśmy robić to w sposób okrężny, badając wspomniane tablice przeglądowe. Naszą implementację można rozszerzyć na przypadek wielu węzłów docelowych, należy wówczas uczynić je wszystkie wartością początkową kolejki *granica_B* i tablicy *osiągnięte_B*.

```
  // węzeł reprezentujący stan końcowy
  węzeł_B ← NODE(problem_B.INITIAL)
  granica_F ← kolejka priorytetowa z funkcją ewaluacyjną f_F,
                początkowo z węzłem węzeł_F jako jedynym elementem
  granica_B ← kolejka priorytetowa z funkcją ewaluacyjną f_B,
                początkowo z węzłem węzeł_B jako jedynym elementem

  osiągnięte_F ← tabela przeglądowa, początkowo z jedną pozycją
                   (węzeł_F.STATE, węzeł_F)
  osiągnięte_B ← tabela przeglądowa, początkowo z jedną pozycją
                   (węzeł_B.STATE, węzeł_B)
  rozwiązanie ← niepowodzenie

  while not TERMINATED(rozwiązanie, granica_F, granica_B) do
    if f_F(TOP(granica_F)) < f_B(TOP(granica_B)) then
      rozwiązanie ← PROCEED(F, problem_F, granica_F,
                osiągnięte_F, osiągnięte_B, rozwiązanie)
    else rozwiązanie ← PROCEED(B, problem_B, granica_B,
              osiągnięte_B, osiągnięte_F, rozwiązanie)
    return rozwiązanie

function PROCEED(kierunek, problem, granica, osiągnięte,
         osiągnięte2, rozwiązanie) returns rozwiązanie
  // Rozwinięcie czołowego węzła granicy; konfrontacja z inną granicą w osiągnięte2.
  // Zmienna kierunek określa kierunek wyszukiwania:
          F oznacza wyszukiwanie w przód, B wyszukiwanie wstecz.
  węzeł ← POP(granica)
  for each potomny in EXPAND(problem, węzeł) do
    s ← potomny.STATE
    if s not in osiągnięte or PATH-COST(potomny) < PATH-
                COST(osiągnięte[s])
    then
      osiągnięte[s] ← potomny
      dodaj potomny do granica
      if s in osiągnięte2 then
        rozwiązanie2 ← JOIN-NODES(kierunek, potomny,
                       osiągniete2[s]))
        if PATH-COST(rozwiązanie2) < PATH-COST(rozwiązanie)
        then
          rozwiązanie ←rozwiązanie2
  return rozwiązanie
```

3.4.6. Porównanie algorytmów wyszukiwania niedoinformowanego

Porównanie algorytmów wyszukiwania niedoinformowanego pod kątem czterech kryteriów omawianych w sekcji 3.3.4 przedstawiamy w tabeli 3.1. Porównanie to dotyczy drzewiastych wersji wyszukiwania, czyli wersji bez sprawdzania powtarzających się stanów. Podstawowa różnica między nimi, a algorytmami w wersji grafowej (które taką kontrolę wykonują) polega na tym, że te ostatnie są zupełne w skończonych przestrzeniach stanów, a ich złożoność — czasowa i pamięciowa — jest proporcjonalna do rozmiaru przestrzeni (mierzonego sumą liczby wierzchołków i krawędzi grafu $|V|+|E|$).

TABELA 3.1. Porównanie podstawowych cech algorytmów wyszukiwania: b oznacza czynnik rozgałęzienia, m jest maksymalną głębokością węzła w drzewie wyszukiwawczym, d jest głębokością najpłytszego rozwiązania, λ jest ograniczeniem głębokości.

Kryterium	Wyszukiwanie wszerz	Wyszukiwanie przy jednolitym koszcie	Wyszukiwanie w głąb	Wyszukiwanie z ograniczeniem głębokości	Iteracyjne zagłębianie	Dwukierunkowość (jeśli możliwa)
Zupełność	Tak[1]	Tak[1, 2]	Nie	Nie	Tak[1]	Tak[1,4]
Optymalność kosztowa	Tak[3]	Tak	Nie	Nie	Tak[3]	Tak[3,4]
Złożoność czasowa	$O(b^d)$	$O\left(b^{1+\lfloor C^*/\varepsilon \rfloor}\right)$	$O(b^m)$	$O\left(b^\lambda\right)$	$O(b^d)$	$O\left(b^{d/2}\right)$
Złożoność pamięciowa	$O(b^d)$	$O\left(b^{1+\lfloor C^*/\varepsilon \rfloor}\right)$	$O(bm)$	$O(b\lambda)$	$O(bd)$	$O\left(b^{d/2}\right)$

[1] Zupełny, jeśli b jest skończone, a przestrzeń stanów zawiera rozwiązanie lub jest skończona.

[2] Zupełny, jeśli koszt każdej akcji jest większy od pewnego dodatniego ε.

[3] Optymalny kosztowo, jeśli wszystkie akcje mają taki sam koszt.

[4] Jeśli w obu kierunkach prowadzone jest wyszukiwanie wszerz lub wyszukiwanie przy jednolitym koszcie.

3.5. Strategie wyszukiwania poinformowanego (heurystycznego)

W tym podrozdziale pokażemy, jak działa strategia wy**szukiwania poinformowanego** (ang. *informed search*), zwanego także **wyszukiwaniem sterowanym wiedzą** — wykorzystując zewnętrzne, specyficzne dla problemu wskazówki dotyczące lokalizacji celów, może ona znajdować rozwiązania bardziej efektywnie niż strategia niedoinformowana. Wspomniane „wskazówki" mają formę **funkcji heurystycznej**, oznaczanej przez $h(n)$ i mającej następujące znaczenie[14]:

> $h(n)$ to szacowany koszt najtańszej ścieżki prowadzącej od stanu reprezentowanego przez węzeł n do stanu docelowego.

[14] Może się wydawać dziwne, że argumentem funkcji heurystycznej jest *węzeł*, skoro jej zadaniem jest estymowanie odległości między *stanami*; otóż notacja $h(n)$ zamiast $h(s)$ podyktowana jest tradycją, a konkretnie względami zgodności z funkcjami ewaluacyjnymi $f(n)$ i funkcjami kosztu ścieżki $g(n)$.

Na przykład, w problemie znajdowania trasy odległość między stanem bieżącym a stanem docelowym może być przybliżana przez *odległość w linii prostej* między punktami na mapie reprezentującymi miasta odpowiadające tym stanom. Heurystykami i ich genezą zajmiemy się dokładniej w podrozdziale 3.6.

3.5.1. Zachłanne wyszukiwanie „najpierw najlepszy"

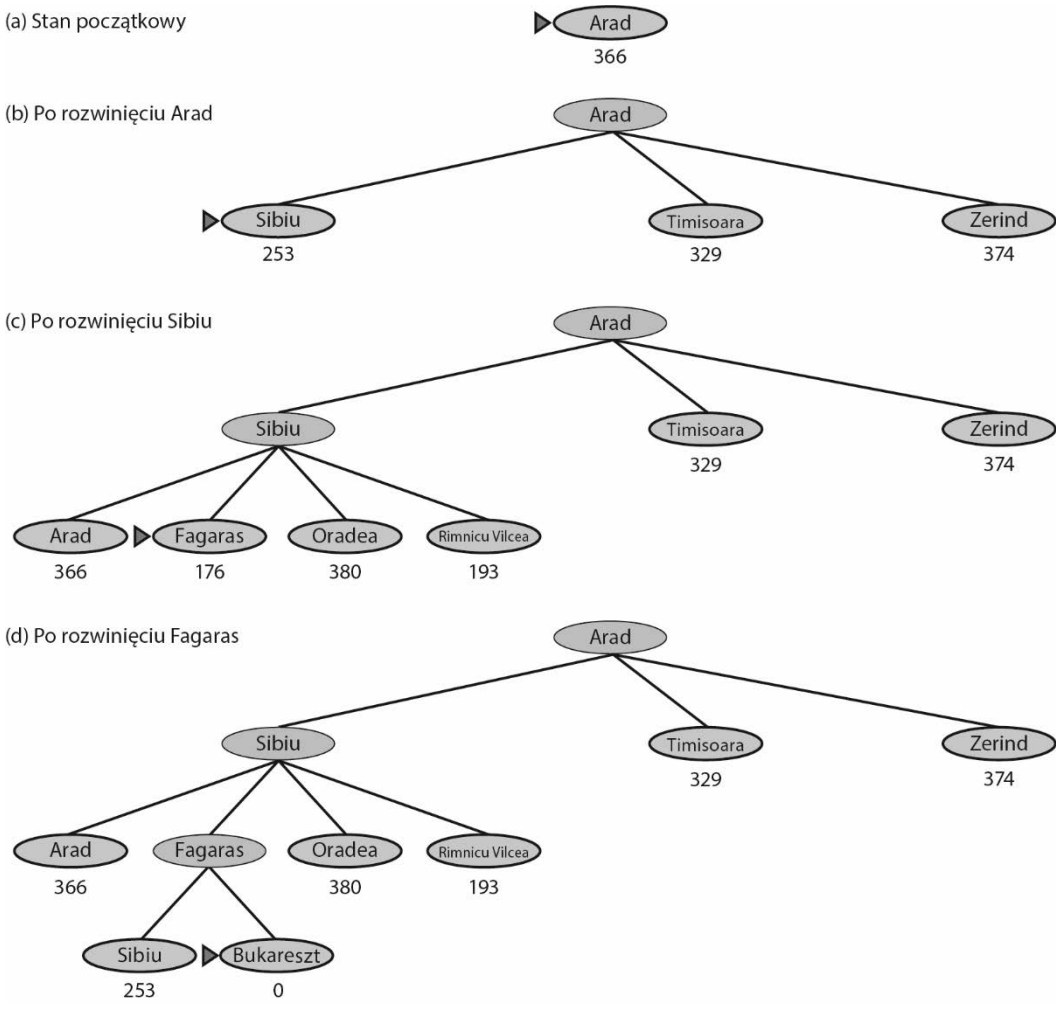
Algorytm wyszukiwania „najpierw najlepszy" w wersji **zachłannej** (ang. *greedy*) wybiera do rozwijania ten węzeł granicy, który wydaje się być położony najbliżej (w sensie funkcji $h(n)$) węzła docelowego — bo prawdopodobnie taka strategia powinna szybko doprowadzić do rozwiązania; innymi słowy, funkcja h jest w tym przypadku funkcją ewaluacyjną ($f(n) = h(n)$).

Zobaczmy, jak strategia ta sprawdza się w przypadku problemu poszukiwania drogi w Rumunii, gdzie jako wartości funkcji heurystycznej przyjmujemy odległość węzła od celu w linii prostej, dlatego oznaczymy tę funkcję przez h_{SLD}, od ang. *straight-line distance*. Wartości tej funkcji dla poszczególnych miast — czyli ich odległości w linii prostej (w milach) od Bukaresztu — zebrane są w tabeli 3.2, na przykład $h_{\text{SLD}}(Arad) = 366$. Zauważmy, że wartości tych nie sposób wydedukować ze specyfikacji problemu, czyli nie można ich wyliczyć przy użyciu funkcji `ACTIONS` i `RESULT`. Zauważmy także, iż użyteczność takiej właśnie heurystyki wynika z faktu, że odległość w linii prostej między miastami skorelowana jest ściśle z ich odległością liczoną wzdłuż dostępnej drogi.

TABELA 3.2. Wartości funkcji h_{SLD} — odległości poszczególnych miast od Bukaresztu, w linii prostej

Arad	366	Mehadia	241
Bukareszt	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Postęp opisanego wyszukiwania zilustrowany jest na rysunku 3.11. Węzłem początkowym (korzeniem) jest *Arad*; jego węzłem potomnym, który zostanie rozwinięty jako pierwszy, jest *Sibiu*, ponieważ (w świetle przyjętej heurystyki h_{SLD}) znajduje się on bliżej Bukaresztu niż węzły *Zerind* i *Timisoara*. Przechodzimy więc do węzła *Sibiu* i (zgodnie z heurystyką) wybieramy do rozwinięcia węzeł *Fagaras*. Na tej samej zasadzie następnym do rozwinięcia jest docelowy węzeł *Bukareszt*. W tym konkretnym przypadku rozwiązanie znalezione zostało bez potrzeby jakiegokolwiek węzła nieznajdującego się na ścieżce będącej rozwiązaniem. Znaleziona trasa *Arad – Sibiu – Fagaras – Bukareszt* nie jest jednak optymalna, bo trasa *Arad – Sibiu – Rimnicu Vilcea – Pitesti – Bukareszt* jest od niej krótsza o 32 mile. Przymiotnik „zachłanny" dla opisanej strategii oznacza, że w każdym kroku stara się ona zbliżyć jak najbardziej do celu; niniejszy przykład jest dowodem na to, że zachłanność nie zawsze popłaca, czyli nie zawsze prowadzi do rozwiązania optymalnego.



RYSUNEK 3.11. Kolejne etapy znajdowania drogi z Arad do Bukaresztu metodą zachłannego wyszukiwania drzewiastego „najpierw najlepszy" przy zastosowaniu heurystyki h_{SLD}. Dla poszczególnych węzłów podano wartości tej heurystyki

Wyszukiwanie grafowe „najpierw najlepszy" w wersji zachłannej jest algorytmem zupełnym w skończonych przestrzeniach stanów, ale nie w nieskończonych. Jego złożoność czasowa i pamięciowa wynosi w najgorszym przypadku $O(|V|)$; jednak przy wyborze odpowiedniej heurystyki można ją znacząco zredukować, w przypadku niektórych problemów nawet do $O(bm)$.

3.5.2. Wyszukiwanie A*

Najbardziej rozpowszechnionym algorytmem wyszukiwania poinformowanego jest **A*** („A z gwiazdką", ang. *A-star*). Jest to w istocie wyszukiwanie „najpierw najlepszy" z funkcją ewaluacyjną określoną jako

$$f(n) = g(n)+h(n)$$

gdzie $g(n)$ jest kosztem ścieżki prowadzącej od węzła początkowego do węzła n, natomiast $h(n)$ oznacza szacowany koszt najkrótszej ścieżki prowadzącej od węzła n do węzła docelowego. W konsekwencji mamy więc

$f(n)$ = szacowany koszt najlepszej ścieżki biegnącej od węzła początkowego, przez węzeł n, do węzła docelowego.

Na rysunku 3.12 pokazujemy postęp wyszukiwania A* jako znajdowania ścieżki z Arad do Bukaresztu. Wartości funkcji $g(n)$ można łatwo obliczyć na podstawie kosztów akcji podanych na rysunku 3.1, wartości funkcji $h(n)$ podano explicite w tabeli 3.2. Zauważmy, że *Bukareszt* po raz pierwszy pojawia się na granicy w kroku (e), ale nie jest wybierany do rozwinięcia (i tym samym nie jest brany pod uwagę jako rozwiązanie), ponieważ przy wartości $f = 450$ nie jest to najtańszy węzeł w granicy — powinien być nim *Pitesti* ($f = 417$). Można to zinterpretować w ten sposób, że może istnieć rozwiązanie prowadzące przez Pitesti, kosztujące niewiele więcej niż 417, więc algorytm nie zadowoli się rozwiązaniem, którego koszt wynosi 450. Gdy w kroku (f) rozwiniemy węzeł *Pitesti*, najtańszym z wygenerowanych węzłów potomnych jest Bukareszt ($f = 418$), jest więc wykrywany i zwracany jako rozwiązanie optymalne.

Wyszukiwanie A* jest algorytmem zupełnym[15]. To, czy jest ono optymalne kosztowo, zależy od pewnych własności przyjętej heurystyki, z których najważniejszą jest **dopuszczalność** (ang. *admissibility*): **heurystyka dopuszczalna** to taka, która nigdy nie przeszacowuje kosztu osiągnięcia celu, czyli jest heurystyką *optymistyczną*. Wyszukiwanie A* z heurystyką dopuszczalną jest optymalne kosztowo, co możemy udowodnić metodą nie wprost, czyli przez sprowadzenie do sprzeczności. Załóżmy mianowicie, że koszt optymalnej ścieżki wynosi C^*, ale algorytm zwraca jako rozwiązanie ścieżkę o koszcie $C > C^*$. Aby tak się stało, musi na optymalnej ścieżce istnieć jakiś *nierozwinięty* węzeł n — gdyby wszystkie węzły na ścieżce optymalnej zostały rozwinięte, algorytm właśnie ją zwróciłby jako rozwiązanie. Oznaczając przez $g^*(n)$ koszt odcinka ścieżki optymalnej od węzła początkowego do węzła n, a przez $h^*(n)$ koszt odcinka ścieżki optymalnej od węzła n do najbliższego węzła docelowego, otrzymujemy:

$f(n) > C^*$ (bo w przeciwnym razie węzeł n musiałby być rozwinięty)

$f(n) = g(n)+h(n)$ (z definicji)

$f(n) = g^*(n)+h(n)$ (bo n znajduje się na optymalnej ścieżce)

$f(n) \le g^*(n)+h^*(n)$ (bo dla heurystyki dopuszczalnej zachodzi $h(n) \le h^*(n)$)

$f(n) \le C^*$ (bo z definicji $C^* = g^*(n)+h^*(n)$)

Jak widać, pierwsza i ostatnia z powyższych równości przeczą sobie nawzajem, czego powodem jest przyjęcie założenia, że algorytm wyszukiwania A* z heurystyką dopuszczalną może zwrócić nieoptymalną ścieżkę jako rozwiązanie.

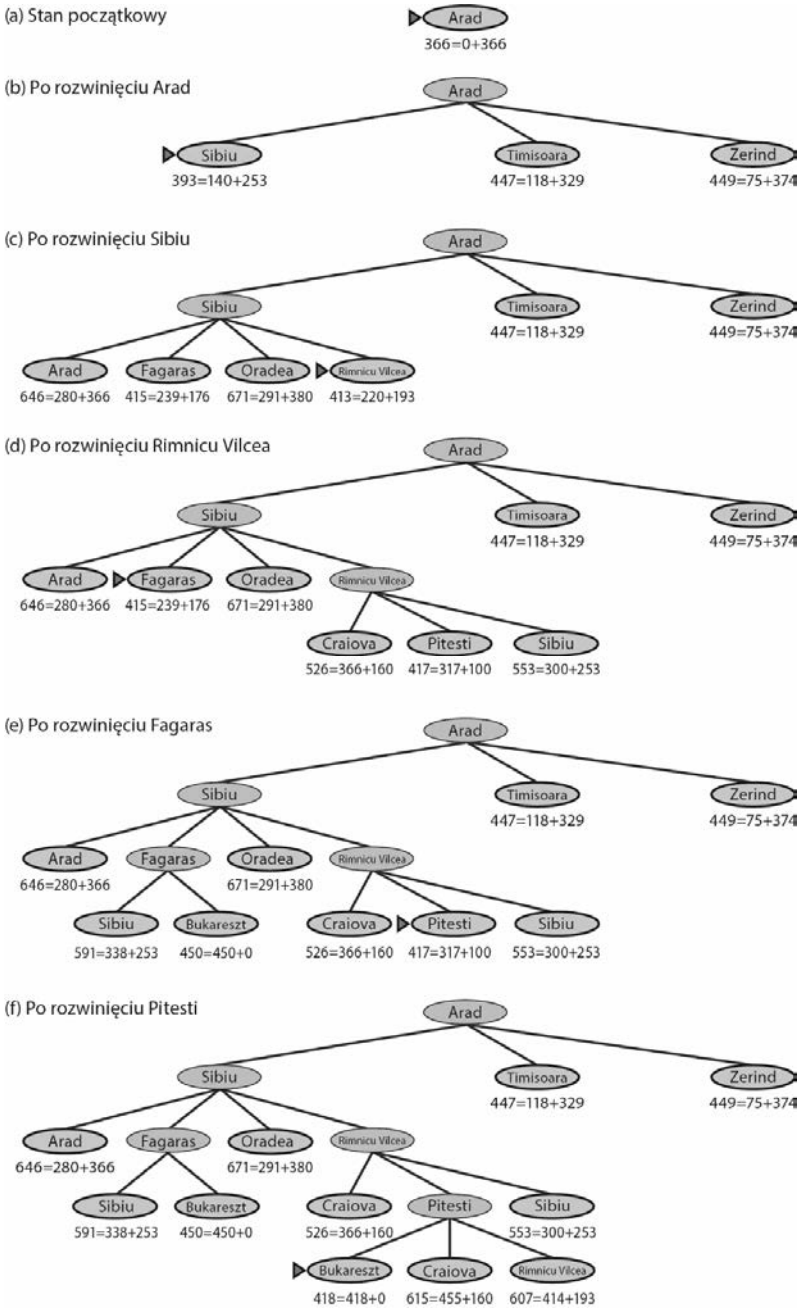
Drugą istotną, nieco silniejszą własnością heurystyki jest jej **spójność** (ang. *consistency*): heurystyka jest **spójna**, jeżeli dla każdego węzła n i każdego jego następnika n' generowanego przez akcję a zachodzi

$$h(n) \le c(n, a, n')+h(n')$$

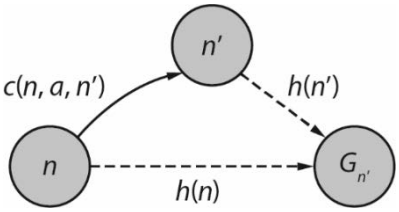
(powyższy warunek jest przykładem **nierówności trójkąta**, na mocy której długość boku trójkąta nie może być większa od sumy długości dwóch pozostałych jego boków[16] (co ilustruje rysunek 3.13). Definiowana wcześniej heurystyka h_{SLD} jest przykładem spójnej heurystyki.

[15] Ponownie zakładamy, że koszt każdej akcji jest większy od pewnego dodatniego ε, a przestrzeń stanów jest skończona lub zawiera rozwiązanie.

[16] Klasyczna nierówność trójkąta jest nierównością *ostrą* — długość dowolnego boku jest *mniejsza* od sumy długości dwóch pozostałych; jeśli dopuścimy nierówność nieostrą, zgadzamy się traktować jako „trójkąty" układy trzech współliniowych punktów — *przyp. tłum.*



RYSUNEK 3.12. Kolejne etapy wyszukiwania A* znajdującego drogę do Bukaresztu. Dla poszczególnych węzłów podano wartości $f = g+h$, gdzie h jest odległością w linii prostej od celu, odczytaną z tabeli 3.2



RYSUNEK 3.13. Nierówność trójkąta ilustrująca spójność heurystyki: dla węzła *n* i jego następnika *n'* generowanego przez akcję *a* wartość $h(n)$ nie może być większa od sumy $h(n')$ i kosztu akcji *a*

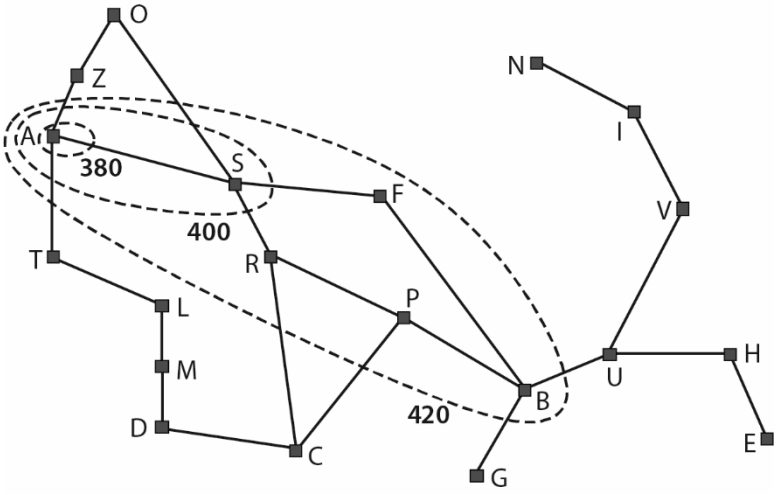
Każda spójna heurystyka jest dopuszczalna (ale nie odwrotnie), więc przy spójnej heurystyce wyszukiwanie A* jest optymalne kosztowo. Ponadto, przy spójnej heurystyce, gdy po raz pierwszy osiągniemy jakiś stan położony na optymalnej ścieżce, nigdy nie będziemy ponownie dodawać tego stanu do granicy ani zmieniać odpowiadającej mu pozycji w tabeli przeglądowej stanów osiągniętych. Nie jest to prawdą w przypadku heurystyki niespójnej — do danego stanu może prowadzić wiele ścieżek, wskutek czego każdorazowo po napotkaniu ścieżki prowadzącej do tego stanu, tańszej niż poprzednia, będziemy ponownie dodawać do granicy węzeł reprezentujący ten stan i modyfikować wspomnianą wcześniej tabelę przeglądową, co wymaga dodatkowego czasu i pamięci. Z tego względu niektóre implementacje A* unikają, w przypadku znalezienia tańszej ścieżki, powtórnego dodawania tego samego stanu do granicy, w zamian modyfikując jego węzły potomne — co oczywiście wymaga utrzymywania, w strukturach reprezentujących węzły, wskaźników zarówno do węzłów potomnych, jak i węzłów macierzystych. Komplikacje te skłaniają autorów algorytmów A* do unikania niespójnych heurystyk, choć Falner i inni (2011) starają się rozwiewać obawy w tym względzie, twierdząc że opisane niepożądane efekty raczej rzadko zdarzają się w praktyce.

Jeśli heurystyka nie jest dopuszczalna, algorytm A* może nie być optymalny kosztowo, ale jest taki przy spełnieniu pewnych warunków. Po pierwsze, jeśli istnieje choćby jedna ścieżka optymalna kosztowo, i dla każdego węzła *n* na tej ścieżce wartość $h(n)$ nie jest zawyżona, to ścieżka ta zostanie zwrócona jako rozwiązanie, bez względu na wartość *h* dla węzłów leżących poza tą ścieżką. Po drugie, jeśli optymalna ścieżka ma koszt C^* i znaleziona zostanie inna ścieżka o koszcie $C_2 > C^*$, to nawet jeśli heurystyka zawyża koszty, ale nie więcej niż o $C_2 - C^*$, istnieje gwarancja, że zwrócone zostanie rozwiązanie o koszcie C^*.

3.5.3. Kontury wyszukiwania

Użytecznym sposobem wizualizacji wyszukiwania są **kontury** naniesione na graf przestrzeni stanów, stanowiące analogię poziomic (warstwic) na mapie topograficznej. Podstawa podziału grafu na kontury są wartości funkcji *f* — wewnątrz konturu etykietowanego wartością *w* znajdują się wszystkie wierzchołki *v*, dla których $f(v) = g(v)+h(v) \leq w$. Na rysunku 3.14 widoczne są trzy kontury dla $w = 380$, 400 i 420. Ponieważ algorytm A* wybiera węzły do rozwinięcia w kolejności rosnących kosztów, budowę drzewa wyszukiwawczego można rozpatrywać jako budowanie koncentrycznych kręgów o zwiększającej się wartości *w*.

Koncepcja konturów ma zastosowanie także do wyszukiwania przy jednolitym koszcie, jednak podstawą etykietowania konturów jest wartość funkcji *g*, nie *g*+*h*. W przeciwieństwie do wyszukiwania A*, gdzie kontury wykazują tendencję do rozszerzania się w kierunku celu, w wyszukiwaniu przy jednolitym koszcie rozszerzanie konturów ma charakter izotropowy — koncentryczne okręgi rozszerzają się równomiernie we wszystkich kierunkach. Przy dobrej heurystyce kontury w wyszukiwaniu A* mogą przybierać kształty wydłużonych elips koncentrujących się wokół optymalnej ścieżki.



RYSUNEK 3.14. Mapa Rumunii z rysunku 3.1 z uwidocznionymi konturami dla $f = 380$, $f = 400$ i $f = 420$, z węzłem początkowym Arad. Dla węzłów znajdujących się wewnątrz konturu etykietowanego wartością w zachodzi zależność: $f+g \leq w$

Ponieważ koszty wszystkich akcji są dodatnie, to oczywiście wartość funkcji g rośnie w miarę wydłużania ścieżki — funkcja g jest więc **monotoniczna**[17]. Linie poszczególnych konturów nie przecinają się więc i zawsze można wstawić linię konturu między dwa dowolne węzły na dowolnej ścieżce (pod warunkiem rysowania konturów wystarczająco cienką kreską).

Nie jest natomiast oczywiste, czy monotoniczna jest także funkcja $f = g+h$. Gdy przedłużamy ścieżkę od węzła n do węzła n', koszt tej ścieżki zmienia się z $g(n)+h(n)$ na $g(n)+c(n, a, n')+h(n')$, czyli jest monotoniczny (niemalejący) wtedy i tylko wtedy, gdy $h(n) \leq c(n, a, n')+h(n')$, czyli gdy heurystyka h jest spójna[18]. Należy jednak uwzględnić fakt, że na ścieżce może pojawić się kilka kolejnych węzłów z identyczną wartością $g(n)+h(n)$ — jest tak wtedy, gdy koszt właśnie podjętej akcji równa się spadkowi funkcji f przy przejściu do nowego węzła. We wcześniejszym problemie z agentem poruszającym się po gridzie jest tak w sytuacji, gdy komórka docelowa znajduje się w tym samym rzędzie co komórka, w której aktualnie znajduje się agent i agent ten robi jeden krok w kierunku celu — funkcja g zwiększa się wówczas o 1 i o tyle samo zmniejsza się funkcja h. Jeśli C^* jest kosztem optymalnego rozwiązania, możemy stwierdzić, że

- A* rozwija wszystkie węzły, które mogą zostać osiągnięte ze stanu początkowego po ścieżce, na której dla każdego węzła n zachodzi $f(n) < C^*$; takie węzły nazywamy **niewątpliwie rozwiniętymi** (ang. *surely expanded*).
- A* może wobec tego rozwijać niektóre z węzłów znajdujących się w „konturze docelowym" (czyli węzłów, dla których $f = C^*$) przed wybraniem węzła docelowego.
- A* nie rozwija węzłów, dla których $f > C^*$.

[17] W zasadzie powinniśmy napisać „ściśle monotoniczna" w odniesieniu do funkcji rosnącej, bo „monotoniczny" oznacza tyle, co „niemalejący", czyli mogący również zachowywać stałą wartość.

[18] Monotoniczność funkcji h ma więc ścisły związek ze spójnością heurystyki i w rezultacie synonimem „heurystyki spójnej" jest „heurystyka monotoniczna", choć oba te terminy zostały sformułowane niezależnie od siebie (Pearl, 1984).

Wyszukiwanie A* ze spójną heurystyką jest **optymalnie wydajne** w tym sensie, że każdy algorytm, który rozszerza ścieżki wyszukiwania ze stanu początkowego i używa tej samej heurystyki, musi rozwinąć wszystkie węzły, które są niewątpliwie rozwijane przez A* (ponieważ każdy tych węzłów mógłby być częścią optymalnego rozwiązania). Jeśli chodzi o węzły, dla których $f = C^*$, to jeden algorytm może szczęśliwym trafem wybrać węzeł położony na optymalnej ścieżce, inny natomiast może mieć mniej szczęścia i w pierwszej kolejności wybrać inny — ta różnica jest nieistotna z perspektywy definicji optymalnej wydajności.

Wyszukiwanie A* jest wydajne, ponieważ **przycina** (ang. *prunes*) drzewo wyszukiwawcze, czyli eliminuje z niego węzły, które nie są konieczne do znalezienia rozwiązania. Na rysunku 3.12 w części (b) widzimy węzły Timisoara ($f = 447$) i Zerind ($f = 449$). Są one potomkami korzenia i byłyby rozwijane w pierwszej kolejności w algorytmie z jednolitym kosztem i algorytmie wyszukiwania wszerz, ale algorytm A* nie rozwinie ich nigdy, ponieważ wcześniej znajdzie rozwiązanie o koszcie $f = 418$.

Koncepcja przycinania — czyli eliminowania a priori pewnych możliwości, bez ich sprawdzania, jako nieistotnych dla rozwiązania — jest ważna w wielu obszarach sztucznej inteligencji.

To, że wyszukiwanie A* jest zupełne, optymalne kosztowo i optymalnie wydajne spośród wszystkich takich algorytmów, jest zdecydowanie dobrą wiadomością. Niestety, nie oznacza to wcale, że stanowi ono odpowiedź na wszelkie problemy wyszukiwania: złą wiadomością jest to, że w wielu problemach liczba rozwiniętych węzłów rośnie wykładniczo do długości ścieżki będącej rozwiązaniem. Powróćmy do (wielokrotnie cytowanego) przykładu z odkurzaczem i załóżmy tym razem, że odkurzacz ten, znajdując się w jakimś kwadracie, ma możliwość odkurzenia *dowolnego kwadratu, bez przemieszczania się do niego*; koszt takiej akcji równy jest 1. Wobec początkowo zakurzonych N kwadratów możliwych jest 2^N stanów, z których każdy charakteryzuje się pewnym podzbiorem kwadratów już odkurzonych. Każdy z tych stanów znajduje się na ścieżce stanowiącej optymalne rozwiązanie, a ponieważ dla każdego z nich mamy $f < C^*$, wszystkie zostaną przeanalizowane w ramach A*.

3.5.4. Wyszukiwanie satysfakcjonujące: heurystyki niedopuszczalne i ważone A*

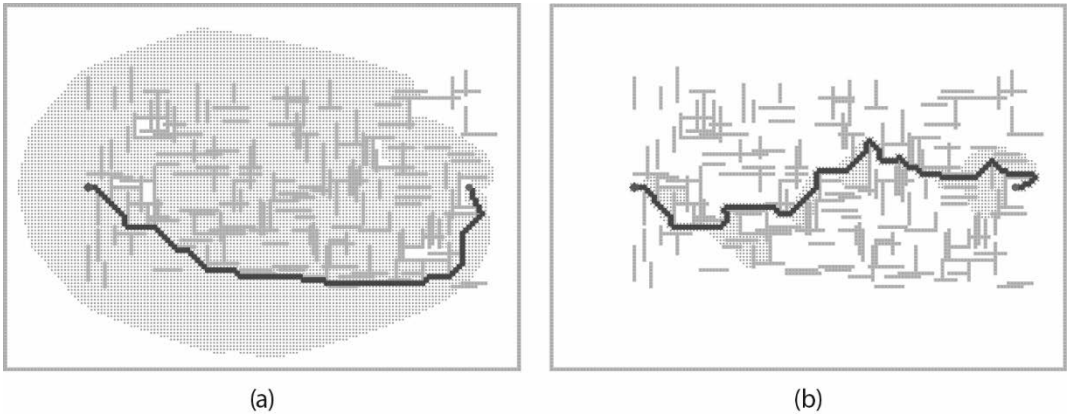
Wyszukiwanie A* ma wiele zalet, ale jego głównym mankamentem jest duża liczba rozwijanych węzłów (dla niektórych problemów). Możemy zredukować liczbę rozwijanych węzłów, i tym samym zaoszczędzić czas i pamięć, jeżeli zrezygnujemy z poszukiwania rozwiązania absolutnie optymalnego i zadowolimy się rozwiązaniem suboptymalnym, które choć nie najlepsze, okazuje się „wystarczająco dobre" w konkretnym zastosowaniu. Takie suboptymalne rozwiązania określamy mianem **satysfakcjonujących** lub **zadowalających**. Jeżeli w wyszukiwaniu A* użyjemy heurystyki, która **nie jest dopuszczalna**, — czyli może zawyżać oszacowania kosztu — ryzykujemy „zgubienie" przez algorytm optymalnego rozwiązania, jednakże heurystyka taka może być potencjalnie dokładniejsza dla danego problemu, co samo z siebie może powodować zredukowanie liczby rozwijanych węzłów. W inżynierii drogowej znane jest pojęcie **współczynnika objazdu** (ang. *detour index*) — jest to stosunek najkrótszej trasy między dwiema lokalizacjami do odległości między nimi w linii prostej; współczynnik ten odzwierciedla stopień „zakrzywienia dróg" w danym regionie. Jeżeli na przykład w danym regionie współczynnik ten wynosi 1,3, a według mapy odległość w linii prostej między dwoma miastami wynosi 10 mil, to można oczekiwać, że jadąc najkrótszą trasą między tymi miastami, trzeba będzie przejechać około 13 mil. Dla większości dróg współczynnik ten waha się w granicach od 1,2 do 1,6.

Tę ideę można uogólnić na dowolny problem — niekoniecznie związany z drogownictwem — w postaci podejścia zwanego **ważonym wyszukiwaniem A*** (ang. *weighted A* search*). Jego istotą jest różnicowanie wkładu funkcji g i h do funkcji f, według formuły

$$f(n) = g(n) + W \times h(n)$$

dla pewnego $W > 1$, zwanego *wagą heurystyki*.

Rysunek 3.15 przedstawia problem wyszukiwania w gridzie. W części (a) widać, jak algorytm A* znajduje optymalne rozwiązanie, badając w tym celu znaczną liczbę węzłów. W części (b) do rozwiązania tego samego problemu użyto ważonego algorytmu A*; otrzymane rozwiązanie ma co prawda koszt trochę większy niż to z części (a), ale znalezione zostało znacznie szybciej. Nietrudno zauważyć, że w wyszukiwaniu ważonym kontury osiąganych celów rozciągnięte są wyraźnie w kierunku celu. Oznacza to, że analizowanych jest znacznie mniej stanów, ale jeśli optymalna ścieżka kiedykolwiek zboczy poza kontury wyszukiwania ważonego (jak w tym przypadku), otrzymane rozwiązanie może nie być rozwiązaniem optymalnym. Ogólnie, jeśli koszt rozwiązania optymalnego wynosi C*, ważone wyszukiwanie znajduje rozwiązanie o koszcie pomiędzy C^* a $W \times C^*$, przy czym w praktyce jest ono położone znacznie bliżej C^* niż $W \times C^*$.



RYSUNEK 3.15. Rezultaty dwóch wyszukiwań na tym samym gridzie: (a) klasyczne wyszukiwanie A* (b) ważone wyszukiwanie A* z wagą $W = 2$. Szare paski to przeszkody między komórkami, purpurowa linia to ścieżka od stanu początkowego (zielony punkt) do stanu docelowego (czerwony punkt). Małe kropki reprezentują stany osiągane w procesie wyszukiwania. W tym konkretnym przypadku ważone wyszukiwanie A* przeanalizowało 7-krotnie mniej stanów i zwróciło rozwiązanie droższe o 5% w porównaniu z klasycznym wyszukiwaniem A*.

Wnikliwi Czytelnicy zauważyli zapewne, że ważone wyszukiwanie A* jest algorytmem dość ogólnym i niektóre inne opisywane w tym rozdziale algorytmy można uważać za szczególne przypadki kombinacji $f = g + W \times h$, dla różnych wartości W, jak w tabeli 3.3.

TABELA 3.3. Algorytmy wyszukiwania jako szczególne przypadki ważonego wyszukiwania A*

Algorytm	Funkcja ewaluacyjna	Waga
Klasyczne wyszukiwanie A*	$g(n)+h(n)$	$W = 1$
Wyszukiwanie przy jednolitym koszcie	$g(n)$	$W = 0$
Zachłanne wyszukiwanie „najpierw najlepszy”	$h(n)$	$W = \infty$
Ważne wyszukiwanie A*	$g(n)+W \times h(n)$	$1 < W < \infty$

Ważone wyszukiwanie A* ma w sobie element zachłanności — przypomina zachłanny algorytm „najpierw najlepszy" w tym sensie, że koncentruje się na podążaniu do celu; nie ignoruje jednak całkowicie kosztu ścieżki i może zawiesić podążanie nią, jeśli zbyt mały postęp okupiony jest zbyt dużymi kosztami.

Istnieje wiele suboptymalnych algorytmów wyszukiwania, które można określić jako „wystarczająco dobre" i scharakteryzować za pomocą kryteriów określających, co jest uważane za „wystarczająco dobre" w świetle różnych kryteriów. **Wyszukiwanie z ograniczoną nieoptymalnością** (ang. *bounded suboptimal search*) daje gwarancję, że koszt zwróconego rozwiązania będzie wyższy od kosztu optymalnego rozwiązania o nie więcej niż stały czynnik ($c \leq W \times C^*$ dla ustalonego $W \geq 1$); wyszukiwanie A* jest właśnie algorytmem tej kategorii. Po **wyszukiwaniu z ograniczeniami kosztu** (ang. *bounded-cost search*) spodziewamy się otrzymać rozwiązanie, którego koszt jest nie większy od ustalonego limitu C. Wreszcie, gdy czynnikiem krytycznym jest czas znajdowania rozwiązania, akceptowalny staje się każdy algorytm **wyszukiwania bez ograniczenia kosztu** (ang. *inbounded-cost search*), o ile gwarantuje zwrócenie rozwiązania w krótkim czasie.

Przykładem algorytmu tej ostatniej kategorii jest **szybkie wyszukiwanie** (ang. *speedy search*), stanowiące odmianę zachłannego wyszukiwania „najpierw najlepszy"; w tym przypadku funkcja heurystyczna dla danego węzła zwraca szacowaną *liczbę akcji* niezbędnych do osiągnięcia celu z poziomu tegoż węzła, niezależnie od kosztu tych akcji. W razie wystąpienia problemów, w których wszystkie akcje mają identyczny koszt, algorytm ten jest identyczny z oryginalną wersją zachłannego wyszukiwania „najpierw najlepszy", w której jako wartość heurystyki przyjmuje się *koszt* osiągnięcia celu.

3.5.5. Wyszukiwanie z ograniczeniami pamięciowymi

Podstawowym mankamentem wyszukiwania A* jest jego pamięciożerność. W tej sekcji omówimy zarówno kilka trików implementacyjnych, dzięki którym udaje się zmniejszyć zapotrzebowanie na pamięć, jak i kilka zupełnie nowych algorytmów, które z natury korzystają z pamięci bardziej oszczędnie.

Główne źródła zapotrzebowania na pamięć to kolejka przechowująca węzły tworzące *granicę* oraz tabela przeglądowa ewidencjonująca stany *osiągnięte*. W przedstawionej przez nas implementacji wyszukiwania „najpierw najlepszy" (patrz listing 3.1) każdy węzeł wchodzący w skład granicy ewidencjonowany jest dwukrotnie: raz w kolejce implementującej tę granice (dzięki której możemy wybierać węzły do rozwijania) i drugi raz w tablicy stanów osiągniętych (dzięki której wiemy, czy już wcześniej analizowaliśmy dany stan). W wielu przypadkach (takich, jak problem eksploracji gridu) dublowanie to nie jest problemem, bo rozmiar *granicy* jest znacznie mniejszy niż rozmiar tablicy stanów *osiągniętych*; w innych przypadkach pożądane jest zredukowanie (w miarę możliwości) zapotrzebowania implementacji algorytmu na pamięć, za pomocą wspomnianych na wstępie trików — ze świadomością, iż skomplikuje to kod i (prawdopodobnie) wydłuży czas poszukiwania rozwiązania.

Jeden z takich trików polega na „przerzedzaniu" tablicy stanów osiągniętych, czyli usuwaniu z niej węzłów, co do których mamy pewność, że nie są już potrzebne; zwolnione w ten sposób pozycje mogą być ponownie wykorzystywane. W przypadku niektórych problemów możemy wykorzystywać własność separacji (patrz rysunek 3.6) w połączeniu z zakazem wykonywania akcji powodujących zawracanie na ścieżce (ang. *U-turn*) w celu zapewnienia, że każda akcja prowadzi bądź to na zewnątrz granicy, bądź do innego stanu tworzącego granicę. Możemy wówczas ograniczyć się do kontrolowania granicy na okoliczność redundantnych ścieżek i tablica ewidencjonująca stany *osiągnięte* nie będzie nam potrzebna.

W przypadku wielu problemów może być użyteczne kontrolowanie **liczby odwołań** do każdego osiągniętego stanu i usuwanie danego stanu z tabeli przeglądowej, gdy liczba ta *osiągnie* limit wynikający z natury problemu. Na przykład w prostokątnym gridzie każda komórka może być osiągnięta wyłącznie wskutek przemieszczenia się do niej z jednej z czterech komórek sąsiednich, więc do stanu odpowiadającego danej komórce algorytm może się odwołać co najwyżej cztery razy; przy czwartym odwołaniu można ów stan usunąć z tabeli przeglądowej.

Przeanalizujmy zatem pokrótce nowe algorytmy zaprojektowane z myślą o oszczędniejszym wykorzystaniu pamięci.

Istotą **wyszukiwania skupionego**, zwanego także **wyszukiwaniem wiązkowym** (ang. *beam search*), jest redukowanie zapotrzebowania na pamięć poprzez ograniczanie liczebności granic — zamiast utrzymywać komplet węzłów stanowiących wynik rozwinięcia, ograniczamy się tylko do ich podzbioru („wiązki") złożonego z k najlepszych, czyli tych, dla których funkcja ewaluacyjna f zwraca najmniejsze wartości. Wskutek tego zabiegu algorytm traci co prawda własność zupełności i optymalności kosztowej, ale za to zmniejsza się jego zapotrzebowanie na pamięć (co może nawet warunkować samą możliwość jego uruchomienia) i oczywiście skraca się czas jego wykonania, bo analizowane jest znacznie mniej stanów. Wybór odpowiedniej wartości k jest swego rodzaju sztuką kompromisu między wymaganiami pamięciowymi a jakością rozwiązania; dla wielu problemów możliwe jest uzyskiwanie rozwiązań bliskich optymalnym, przy jednoczesnej znaczącej redukcji zapotrzebowania na pamięć. Powracając do koncepcji konturów wyszukiwania z sekcji 3.5.3: można sobie wyobrazić wyszukiwanie A* przy jednolitym koszcie jako promieniowanie (ang. *beaming*) *wszystkich* węzłów objętych konturem na zewnątrz tego konturu, a wyszukiwanie skupione — jak promieniowanie tylko „wiązki" k najlepszych kandydatów.

W odmianie tego algorytmu liczebność granicy (k) nie jest sztywno ustalona a priori, wiązkę tworzą te węzły, dla których funkcja ewaluacyjna zwraca wartość różniącą się o nie więcej niż ustalone δ od wartości minimalnej funkcji f. Liczebność podzbioru rozwijanych węzłów zależy zatem od „polaryzacji" granicy, czyli liczby węzłów wyraźnie odróżniających się (w sensie funkcji f) od reszty; nawet gdy liczebność ta jest duża, może gwałtownie zmaleć, gdy pojawi się odpowiednio „silny" węzeł.

Wyszukiwanie A* z iteracyjnym zagłębianiem (ang. IDA* — *Iterative-deepening A∗ search*) ma się tak do klasycznego wyszukiwania A*, jak iteracyjne zagłębianie do wyszukiwania w głąb: korzyści IDA* biorą się z braku konieczności przechowywania w pamięci wszystkich osiągniętych stanów, za cenę wielokrotnego analizowania niektórych stanów. Algorytm ten ma duże znaczenie, i jest powszechnie wykorzystywany, w rozwiązywaniu problemów, których wymagania pamięciowe wykluczają użycie innych algorytmów.

W klasycznym zagłębianiu iteracyjnym limit głębokości powiększany jest o jeden w każdej iteracji; w algorytmie IDA* limit głębokości w kolejnej iteracji wyznaczony zostaje przez węzeł, dla którego wartość funkcji f (równa sumie wartości funkcji g i h) jest najmniejsza i jednocześnie większa od tej wyznaczającej limit w poprzedniej iteracji. Innymi słowy, algorytm IDA* w każdej iteracji dokonuje wyczerpującego przeszukiwania f-konturu, znajduje zewnętrzny węzeł położony najbliżej tego konturu i traktuje funkcję f dla tego węzła jako wartość nowego konturu. W przypadku problemów takich jak puzzle-8 (patrz rysunek 3.3), w których funkcja f przyjmuje tylko wartości całkowite, strategia ta działa znakomicie, konsekwentnie przybliżając rozwiązanie wraz z kolejnymi iteracjami. Jeśli więc koszt optymalnego rozwiązania wynosi C^*, liczba iteracji wynosi co najwyżej C^* (31 w najtrudniejszej konfiguracji puzzle-8); w przypadku, gdy funkcja f zwraca inną wartość (niekoniecznie całkowitą) dla każdego węzła, każdy węzeł wyznacza nowy kontur i liczba iteracji równa jest liczbie stanów.

Rekurencyjne wyszukiwanie „najpierw najlepszy" (ang. RBFS — *Recursive Best-First Search* — listing 3.5) naśladuje klasyczną wersję wyszukiwania „najpierw najlepszy", lecz w przeciwieństwie do niej charakteryzuje się liniowym zapotrzebowaniem na pamięć. RBFS przypomina nieco rekurencyjne wyszukiwanie w głąb, ale zamiast posuwać się w nieskończoność na bieżącej ścieżce, kieruje się wartością zmiennej (`limit_f`) zapamiętującej wartość kosztu (funkcji f) najlepszej *alternatywnej* ścieżki prowadzącej od dowolnego przodka[19] bieżącego węzła. Gdy wartość funkcji f dla bieżącego węzła przekracza wartość zmiennej `limit_f`, rekurencja powraca do ścieżki alternatywnej i cofając się, zastępuje wartość f każdego węzła na ścieżce wartością najlepszą spośród jego węzłów potomnych, **zarchiwizowaną** w zmiennej `limit_f`. W rezultacie RBFS zapamiętuje najlepszą wartość f wśród liści „zapomnianego" poddrzewa i na tej podstawie podejmuje decyzję, czy drzewo to warte jest późniejszego ponownego rozwijania. Rysunek 3.16 pokazuje, jak RBFS znajduje drogę z Arad do Bukaresztu.

[19] „Przodek" to przechodnie domknięcie (niepuste) relacji „węzeł macierzysty". Niech M będzie węzłem macierzystym węzła X; węzeł Y jest przodkiem węzła X, jeśli $Y = M$ lub Y jest przodkiem M. Mówiąc potocznie, przodkami węzła X są: jego węzeł macierzysty, węzeł macierzysty jego węzła macierzystego itd. — *przyp. tłum*.

LISTING 3.5. Algorytm rekurencyjnego wyszukiwania „najpierw najlepszy"

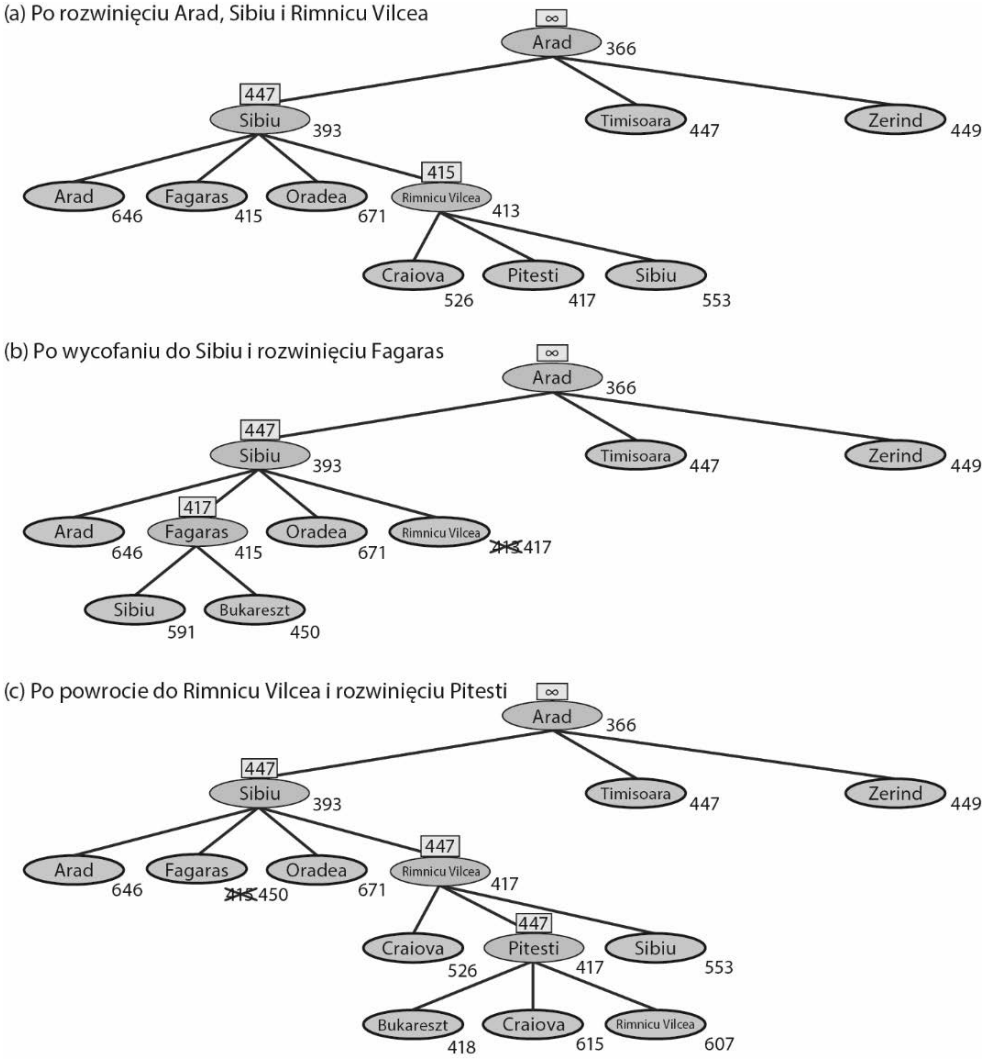
```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns rozwiązanie albo sygnalizacja niepowodzenia
  rozwiązanie, wartość_f ← RBFS(problem, NODE(problem.INITIAL), ∞)
  return rozwiązanie

function RBFS(problem, węzeł, limit_f) returns rozwiązanie albo sygnalizacja niepowodzenia
                                                oraz nowy koszt limit_f
  if problem.IS-GOAL(węzeł.STATE) then return węzeł
  następniki ← LIST(EXPAND(węzeł))
  if następniki jest pustą listą then return niepowodzenie, ∞
  for each s in następniki do // aktualizuj f wartością z poprzedniego wyszukiwania
    s.f ← max(s.PATH-COST + h(s), węzeł.f )

  while prawda do
    najlepszy ← węzeł o najmniejszej wartości f w liście następniki
    if najlepszy.f > limit_f then return niepowodzenie, najlepszy.f
    alternatywna ← druga najmniejsza wartość_f w liście następniki
    wynik, najlepszy.f ← RBFS(problem, najlepszy, min(limit_f, alternatywna))
    if wynik ≠ niepowodzenie then return wynik, najlepszy.f
```

Algorytm RBFS jest nieco wydajniejszy niż IDA*, ale nadal jego bolączką jest nadmierna regeneracja węzłów. W przykładzie na rysunku 3.16, RBFS podąża ścieżką przez Rimnicu Vilcea, by wkrótce rozmyślić się i spróbować drogi przez Fagaras, po czym znowu zmienić zdanie. To swoiste niezdecydowanie bierze się stąd, że gdy aktualnie najlepsza ścieżka jest wydłużana, jej koszt prawdopodobnie rośnie — funkcja *h* jest zwykle mniej optymistyczna dla węzłów położonych blisko celu. W takim przypadku druga co do jakości ścieżka może stać się tą najlepszą, a więc wyszukiwanie musi się cofnąć, by nią podążyć. Każda taka zmiana decyzji odpowiada pojedynczej iteracji IDA* i może wymagać ponownej eksploracji „zapomnianych" węzłów w celu odtworzenia najlepszej ścieżki i rozszerzenia jej o kolejny węzeł.

RBFS jest algorytmem optymalnym kosztowo, o ile heurystyka *h* jest heurystyką dopuszczalną. Jego wymagania pamięciowe rosną proporcjonalnie do najgłębszego optymalnego rozwiązania, natomiast jego złożoność czasowa jest raczej trudna do sprecyzowania, bo zależy (między innymi) od adekwatności funkcji *h* oraz częstotliwości zmian optymalnej ścieżki w miarę rozwijania węzłów. Węzły rozwijane są w kolejności rosnącej funkcji *f*, nawet jeśli nie jest ona monotoniczna.

Oszczędność wykorzystywania pamięci, skądinąd ogólnie bardzo pożądana, w przypadku algorytmów IDA* i RBFS jest poniekąd ich słabą stroną. Pomiędzy kolejnymi iteracjami IDA* przechowywana jest tylko jedna liczba — limit kosztu (funkcji *f*). RBFS zapamiętuje co prawda więcej informacji, ale wykorzystuje pamięć jedynie w stopniu proporcjonalnym („liniowym") do zagłębienia ścieżki i nawet gdyby pamięci było znacznie więcej, to i tak nie może zrobić z niej użytku. Przyrodzona obu algorytmom amnezja co do wcześniejszych działań skazuje je na wielokrotne analizowanie tych samych stanów.

Rozsądne wydaje się zatem żądanie, by można było nakazać algorytmowi wykorzystywanie pamięci w jawnie zadanej wielkości, a być może i całej dostępnej pamięci. Dwa algorytmy, które mają taką możliwość, to **MA*** (ang. *Memory-Bounded A** — A* ograniczone pamięciowo) i **SMA*** (ang. *simplified MA** — uproszczone MA*); opiszemy tu SMA* jako (zgodnie z nazwą) mniej skomplikowany. SMA* działa podobnie jak A*, rozwijając najlepszy liść aż do wyczerpania dostępnej pamięci — i tym samym niemożności dodania nowego węzła do drzewa wyszukiwawczego, bez usunięcia któregoś z istniejących węzłów; SMA* usuwa zatem *najgorszy* liść — ten, dla którego funkcja *f* zwraca największą wartość — i jednocześnie archiwizuje tę wartość w jego węźle macierzystym. Ów węzeł macierzysty „zapomnianego" poddrzewa posiada informację na temat najlepszej ścieżki w tym poddrzewie. Dzięki tej informacji SMA* regeneruje wspomniane poddrzewo tylko wtedy, kiedy inne ścieżki okazują się gorsze od tej „zapomnianej". Innymi słowy, jeśli zapomniane zostaną wszystkie węzły potomne węzła *n*, nie wiadomo, dokąd podążać dalej z tego węzła.



RYSUNEK 3.16. Wyszukiwanie RBFS najkrótszej ścieżki do Bukaresztu. Nad bieżącymi węzłami widoczne są wartości zmiennej `limit_f` zwracanej przez ostatnie wywołanie rekurencyjne, a każdy węzeł etykietowany jest wartością funkcji *f*. Etap (a): ścieżka przez Rimnicu Vilcea jest rozwijana aż do wystąpienia sytuacji, gdy najlepszy liść (Pitesti) ma wartość gorszą od najlepszej ścieżki alternatywnej (Fagaras). Etap (b): rekurencja cofa się i wartość *f* najlepszego liścia „zapomnianego" poddrzewa (417) jest archiwizowana w węźle Rimnicu Vilcea. Następnie rozwijany jest węzeł Fagaras, co prowadzi do wykrycia wartości 450 jako najlepszej wśród liści. Etap (c): rekurencja cofa się i wartość najlepszego liścia „zapomnianego" poddrzewa (450) jest archiwizowana w węźle Fagaras, po czym rozwijany jest węzeł Rimnicu Vilcea. Ponieważ w tym momencie koszt najlepszej ścieżki alternatywnej (przez Timisoara) jest równy co najmniej 447 (czyli więcej niż 417), rozwinięcie węzła Rimnicu Vilcea prowadzi do Bukaresztu

Kompletny opis algorytmu znajduje się w repozytorium online z przykładowym kodem do niniejszej książki, w tym miejscu chcielibyśmy zwrócić uwagę na pewną subtelność. Otóż, jak wyjaśniliśmy, SMA* rozwija *najlepszy* liść, jednocześnie usuwając z pamięci liść *najgorszy* — co się więc stanie, gdy liść najlepszy będzie jednocześnie najgorszym, bo funkcja *f* zwraca j*ednakową wartość* dla wszystkich liści? SMA* rozstrzyga ten problem w ten sposób, że rozwija *najnowszy* z najlepszych liści i usuwa *najstarszy* z najgorszych, co wyklucza możliwość wybrania tego samego węzła jednocześnie do rozwinięcia i do usunięcia. Oczywiście wciąż nie rozwiązuje to problemu w sytuacji, gdy istnieje *tylko jeden liść*, lecz w takim przypadku bieżące drzewo wyszukiwawcze musi mieć formę pojedynczej ścieżki, której węzły zajmują *całą dostępną pamięć*. Jeśli ów samotny liść nie jest celem, to nawet gdy znajduje się on na ścieżce do optymalnego rozwiązania, rozwiązanie to jest nieosiągalne z powodu *braku wolnej pamięci*. Samotny liść zostanie więc usunięty z pamięci dokładnie tak, jak gdyby nie posiadał węzłów potomnych.

SMA* jest algorytmem zupełnym, jeżeli istnieje osiągalne rozwiązanie — czyli gdy najpłytszy węzeł docelowy znajduje się na głębokości *d* takiej, że dostępna pamięć może pomieścić co najmniej *d*+1 węzłów. Algorytm jest optymalny kosztowo, jeżeli osiągalne jest dowolne rozwiązanie optymalne, w przeciwnym razie zwraca on najlepsze spośród osiągalnych rozwiązań. Z praktycznego punktu widzenia SMA* jest dość dobrym wyborem dla znajdowania optymalnych rozwiązań w sytuacji, gdy przestrzeń stanów tworzy graf, akcje mają zróżnicowane koszty, a generowanie węzłów jest kosztowne w porównaniu z kosztami zarządzania granicą i tablicą stanów osiągniętych.

Jednak w przypadku szczególnie trudnych problemów może się zdarzyć, że SMA* zmuszony będzie przełączać się nieustannie między wieloma ścieżkami, stanowiącymi kandydatury na rozwiązanie, i tylko niewielki podzbiór tych ścieżek zmieści się w dostępnej pamięci. Wystąpi wówczas, dobrze znane podsystemom zarządzania pamięcią wirtualną, zjawisko **migotania stron** (ang. *threashing*) polegające na tym, że system komputerowy większość czasu spędza na nieustannym przerzucaniu treści między pamięcią RAM a plikiem wymiany, zamiast wykonywać użyteczne obliczenia. I tak oto *problem teoretycznie rozwiązywalny przy użyciu algorytmu A* i dostępności nieograniczonej pamięci, może okazać się nierozwiązywany ze względu na nieakceptowalnie duży czas wykonania algorytmu wynikający z ograniczeń pamięciowych.* Tak to już jest, że oszczędność pamięci okupiona jest zwiększonym czasem obliczeń (i vice versa) i nie istnieje żadna teoria pozwalająca na wypracowanie kompromisu w tym względzie; opisane zjawisko jest nieuniknione i wydaje się, że jedynym sposobem jego złagodzenia jest rezygnacja z wygórowanych wymagań dotyczących optymalności rozwiązania.

3.5.6. Dwukierunkowe wyszukiwanie heurystyczne

Przy jednokierunkowym wyszukiwaniu „najpierw najlepszy" użycie funkcji ewaluacyjnej $f(n) = g(n)+h(n)$ prowadzi do wyszukiwania A*, gwarantującego znalezienie rozwiązań optymalnych kosztowo (przy założeniu, że *h* jest heurystyką dopuszczalną) przy jednocześnie optymalnej wydajności w sensie liczby generowanych węzłów.

W przypadku wyszukiwania „najpierw najlepszy" w wersji dwukierunkowej moglibyśmy użyć takiej samej funkcji ewaluacyjnej, jednak nie daje to gwarancji ani optymalności kosztowej rozwiązania, ani optymalnej wydajności, nawet jeśli heurystyka *h* jest dopuszczalna. Jest zrozumiałe, że przy dwukierunkowym wyszukiwaniu analizujemy nie pojedyncze węzły, lecz raczej *pary węzłów* (po jednym z każdej granicy) co do których można udowodnić, że są niewątpliwie rozwinięte, więc każdy dowód, że wydajność algorytmu jest optymalna, musi uwzględniać właśnie pary węzłów (Eckerle i in., 2017).

W związku z dwukierunkowością wyszukiwania zdefiniujemy nową notację: oznaczenia związane z kierunkiem w przód opatrywać będziemy indeksem *F*, zaś oznaczenia związane z kierunkiem w tył — indeksem *B*. Zatem funkcja ewaluacyjna dla kierunku w przód będzie miała postać $f_F(n) = g_F(n)+h_F(n)$, a dla kierunku w tył — $f_B(n) = g_B(n)+h_B(n)$. Mimo iż oba kierunki związane są z rozwiązywaniem tego samego problemu, dla każdego z nich obowiązuje inna funkcja ewaluacyjna, chociażby ze względu na różnicę obu heurystyk: jedna odzwierciedla dążenie do węzła docelowego, druga — do węzła początkowego. Zakładamy, że obie heurystyki są dopuszczalne.

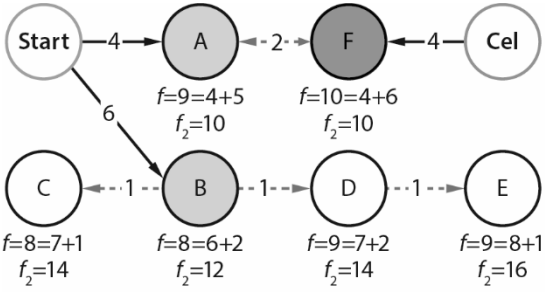
Weźmy pod uwagę dwie ścieżki: jedną od węzła początkowego do węzła m i drugą od węzła docelowego do węzła n. Zdefiniujemy dolne ograniczenie (*lb*, od ang. *lower bound*) kosztu rozwiązania, które podąża najpierw pierwszą ścieżką od węzła początkowego do węzła m, następnie dociera do węzła n i podąża druga ścieżką do węzła docelowego:

$$lb(m,n) = \max(g_F(m)+g_B(n), f_F(m), f_B(n))$$

Innymi słowy, koszt takiej ścieżki musi być co najmniej tak duży, jak suma kosztów g obu ścieżek częściowych (ponieważ koszt odcinka łączącego obie części musi być nieujemny), a ponadto nie może być mniejszy od kosztu f którejkolwiek ze ścieżek częściowych (bo wartości heurystyki są optymistyczne). Można udowodnić, że dla dowolnej pary węzłów (m, n) spełniającej warunek $lb(m, n) < C^*$ musimy rozwinąć oba te węzły, ponieważ przechodząca przez nie ścieżka może być potencjalnym rozwiązaniem optymalnym. Sęk w tym, że nie mamy pewności, który węzeł najlepiej rozwinąć, zatem żaden algorytm wyszukiwania dwukierunkowego nie może gwarantować optymalnej wydajności — w skrajnym wypadku, gdy z każdej pary wybrany zostanie niewłaściwy węzeł do rozwinięcia jako pierwszy, liczba wygenerowanych węzłów może być dwukrotnie większa od minimalnej. Niektóre dwukierunkowe algorytmy wyszukiwania heurystycznego w sposób jawny zarządzają kolejką par (m, n), my jednak pozostaniemy przy wyszukiwaniu „najpierw najlepszy" z listingu 3.4 implementującym granice w postaci kolejek priorytetowych i definiującym funkcję ewaluacyjną zgodnie z kryterium dolnego ograniczenia:

$$f_2(n) = \max(2g(n), g(n)+h(n))$$

Jako następny do rozwinięcia wybierany jest węzeł o najmniejszej wartości f_2 (niezależnie od tego, do której granicy należy). Wyklucza to możliwość wybrania do rozwinięcia węzła n dla którego zachodzi $g(n) > \frac{C^*}{2}$. Mówimy, że dwie połówki wyszukiwania „spotykają się pośrodku" w tym sensie, że w żadnej z granic nie istnieje węzeł, dla którego koszt ścieżki przekraczałby wspomniane $\frac{C^*}{2}$. Przykładowe wyszukiwanie dwukierunkowe zilustrowaliśmy schematycznie na rysunku 3.17.



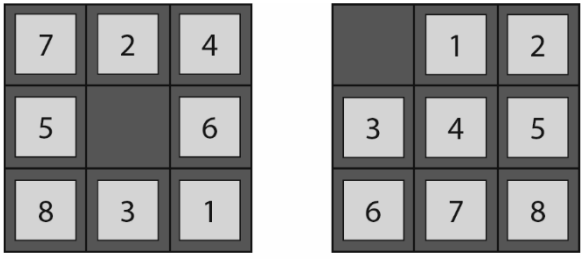
RYSUNEK 3.17. Dwukierunkowe wyszukiwanie utrzymuje dwie kolejki węzłów. Z lewej strony węzły A i B są następnikami węzła początkowego *Start*; z prawej strony węzeł F jest wstecznym następnikiem węzła *Cel*. Dla każdego węzła podano wartość funkcji ewaluacyjnej $f = g+h$ i wartość $f = max(2g, g+h)$. Wartością funkcji g jest suma kosztów akcji, uwidocznionych przy poszczególnych strzałkach, wartości heurystyki h przyjęto dowolnie (nie da się ich wydedukować z rysunku). Koszt optymalnego rozwiązania *Start-A-F-Cel* wynosi $C^* = 4+2+4 = 10$, co oznacza, że dwukierunkowy algorytm „spotkania pośrodku" nie powinien rozwijać żadnego węzła, dla którego $g > \frac{C^*}{2} = 5$, więc następnym rozwijanym węzłem na drodze do optymalnego rozwiązania powinien być A albo F (dla obu $g = 4$). Gdybyśmy kierowali się najmniejszą wartością funkcji f, wybralibyśmy do rozwinięcia węzły B i C ($f = 8$) zamiast A ($f = 9$), następnie D i E zostałyby związane z A. Dla nich wszystkich jednak $g > \frac{C^*}{2}$, więc żaden z nich nie zostanie rozwinięty, gdy funkcją ewaluacyjną będzie f_2

W opisanym algorytmie heurystyka h_F szacuje odległość danego węzła od celu (lub, gdy problem ma wiele stanów docelowych, odległość od najbliższego celu), analogicznie h_B szacuje odległość od węzła początkowego. Nazywa się to **wyszukiwaniem od czoła do celu** (ang. *front-to-end*). Alternatywa, zwana **wyszukiwaniem od czoła do czoła** (ang. *front-to-front*) wykorzystuje heurystykę mierzącą odległość węzła do drugiej granicy. Oczywiście, gdy granica zawiera miliony węzłów, obliczanie heurystyki dla nich wszystkich (w celu znalezienia tego z najmniejszą wartością) byłoby nieefektywne; rozwiązaniem zastępczym jest wówczas ograniczenie się do niewielkiej *próbki węzłów* losowo wybieranych z granicy. W przypadku pewnych specyficznych domen problemowych możliwe jest sumowanie granicy — na przykład w problemie przeszukiwania gridu możemy przyrostowo obliczać „obwiednię" granicy i w charakterze heurystyki przyjmować odległość węzła od tejże obwiedni.

Relacje między wyszukiwaniem dwukierunkowym a jednokierunkowym są różne — dwukierunkowe jest niekiedy bardziej wydajne, niekiedy wręcz przeciwnie. Generalnie, jeśli mamy bardzo dobrze zdefiniowaną heurystykę, to wyszukiwanie A* generuje kontury koncentrujące się w kierunku celu i dodanie wyszukiwania dwukierunkowego niewiele może pomóc. W przypadku przeciętnej heurystyki wyszukiwanie dwukierunkowe ze spotkaniem granic pośrodku cechuje się mniejszą liczbą rozwijanych węzłów i jako takie jest preferowane. W przypadku najgorszym, czyli przy źle zdefiniowanej heurystyce, wyszukiwanie nie wykazuje tendencji koncentrowania się na celu i wariant dwukierunkowy ma taką samą asymptotyczną złożoność jak A*. Wyszukiwanie dwukierunkowe z funkcją ewaluacyjną f_2 i heurystyką dopuszczalną jest algorytmem zupełnym i optymalnym kosztowo.

3.6. Funkcje heurystyczne

W tym podrozdziale postaramy się pokazać, jak adekwatność zdefiniowanej heurystyki warunkuje wydajność wyszukiwania oraz jak w ogóle wynajdywane są funkcje heurystyczne. W charakterze przykładu użyjemy grę puzzle-8, której planszę przedstawiliśmy po raz pierwszy na rysunku 3.3 i reprodukujemy ponownie na rysunku 3.18. Przypomnijmy — na planszy o wymiarach 3×3 można przesuwać (poziomo lub pionowo, ale nie na skos) 8 kwadratowych kafelków o rozmiarach 1×1 każdy, aby otrzymać pewną predefiniowaną konfigurację docelową; najczęściej w tej roli przyjmowana jest konfiguracja widoczna w prawej części rysunku 3.18.



RYSUNEK 3.18. Jedna z typowych instancji gry puzzle-8, najkrótsze rozwiązanie wymaga 26 akcji

Liczba osiągalnych stanów w tej grze wynosi 9!/2 = 181 400 i przechowywanie ich wszystkich w pamięci jest sprawą banalną. Ale w odmianie puzzle-15 — z 15 kafelkami na planszy 4×4 — liczebność przestrzeni stanów sięga 16!/2, czyli ponad 10 bilionów, a jej przeszukiwanie wymaga wsparcia naprawdę dobrej heurystyki dopuszczalnej. Poszukiwanie takiej heurystyki ma długą historię, najczęściej stosowanymi heurystykami są dwie następujące:

- h_1 — równa jest liczbie kafelków znajdujących się na niewłaściwej pozycji; w części (a) rysunku 3.18 żaden z 8 kafelków nie znajduje się na swoim miejscu, zatem $h_1 = 8$. Heurystyka ta jest dopuszczalna, ponieważ każdy niewłaściwie ulokowany kafelek wymaga co najmniej jednego ruchu w celu umieszczenia go na właściwym miejscu.
- h_2 — równa jest sumarycznej odległości dzielącej kafelki od ich pozycji docelowych; ponieważ kafelki nie mogą poruszać się po przekątnej, więc wspomniana „odległość" dla każdego kafelka równa jest sumie ruchów poziomych i pionowych koniecznych do jego sprowadzenia na docelową pozycję. Ograniczenie możliwości ruchu do dwóch prostopadłych kierunków przypomina układ ulic w dużych miastach, liczona w ten sposób odległość nosi nazwę **odległości miejskiej** (ang. *city-block distance*), osiedlowej lub **odległości Manhattanu** (ang. *Manhattan distance*). Gdy zsumujemy odległości kafelków w części (a) rysunku 3.18, otrzymamy

  $h_2 = 3+1+2+2+2+3+3+2= 18$

Jak można się było spodziewać, żadna z tych heurystyk nie zawyża rzeczywistego kosztu rozwiązania, który równy jest 26.

3.6.1. Dokładność heurystyki a wydajność algorytmu

Jednym z kryteriów oceny jakości heurystyki jest **efektywny czynnik rozgałęzienia**. W drzewie jednorodnym (czyli takim, w którym każdy węzeł niebędący liściem ma dokładnie b węzłów potomnych) o głębokości d liczba węzłów jest równa

$$W = 1 + b + b^2 + b^3 + \cdots + b^d$$

Tę zależność możemy uogólnić na dowolne drzewo wyszukiwawcze. Jeśli w procesie wyszukiwania A* generowanych jest N węzłów, a najgłębszy węzeł znajduje się na głębokości d, to istnieje wartość b^* spełniająca równanie

$$N + 1 = 1 + b^* + (b^*)^2 + (b^*)^3 + \cdots + (b^*)^d$$

(dodajemy 1 do N, by uwzględnić węzeł początkowy). Ta właśnie wartość to wspomniany efektywny czynnik rozgałęzienia.

I tak na przykład, jeśli algorytm A* znajdzie rozwiązanie o głębokości 5, generując 42 węzły, to wartość b^* równa jest około 1,92. Efektywny czynnik rozgałęzienia może być różny w różnych instancjach problemu, ale zwykle dla określonej domeny (na przykład gry puzzle-8) jest mniej więcej taki sam we wszystkich niebanalnych instancjach problemu. Wyniki eksperymentalnych pomiarów czynnika b^* na małym zbiorze problemów pewnej domeny mogą zatem stanowić dobry przewodnik dotyczący użyteczności zastosowanej heurystyki. Przy dobrze zaprojektowanej heurystyce współczynnik ten powinien mieć wartość bliską 1, co oznacza, że nawet złożone problemy mogą być rozwiązywane kosztem rozsądnej mocy obliczeniowej.

Korf i Reid (1998) twierdzą natomiast, że lepszym miernikiem obcinania w algorytmie A* przy zastosowaniu określonej heurystyki h jest czynnik k_h, o jaki rzeczywista głębokość d drzewa wyszukiwawczego zredukowana zostaje do **głębokości efektywnej** d_e — w tym sensie, że całkowity koszt wyszukiwania zmniejsza się z $O(bd)$ do

$$O(b^{d_e}) = O(b^{d-k_h})$$

Ich eksperymenty z kostką Rubika i puzzlami-k (dla różnych wartości k) pokazują, że powyższa formuła umożliwia trafne przewidywanie całkowitego kosztu wyszukiwania dla badanych instancji problemów w szerokim zakresie długości zwracanych rozwiązań — przynajmniej tych większych od k_h.

W tabeli 3.4 przedstawiamy rozwiązania losowo wygenerowanych instancji problemu układanki puzzle-8, przy użyciu algorytmu BFS i algorytmu A* z (opisywanymi wcześniej) heurystykami h_1 i h_2. Dla poszczególnych długości rozwiązania podajemy średnią liczbę wygenerowanych węzłów i wartość efektywnego czynnika rozgałęzienia. Wyniki te wykazują zarówno przewagę heurystyki h_2 nad h_1, jak i wyraźną korzyść stosowania heurystyki w ogóle.

TABELA 3.4. Porównanie kosztu wyszukiwania i efektywnego czynnika rozgałęzienia dla losowo wygenerowanych instancji problemu puzzle-8, przy użyciu algorytmów: „najpierw najlepszy" oraz A* z dwiema heurystykami: h_1 (zwracającą liczbę niewłaściwie ulokowanych kafelków) i h_2 (zwracającą sumaryczna odległość Manhattanu dla kafelków). Wyniki zostały uśrednione po 100 instancjach problemu dla każdej długości rozwiązania od 6 do 28

Długość rozwiązania	**Koszt wyszukiwania (liczba wygenerowanych węzłów)**			**Efektywny czynnik rozgałęzienia**		
	BFS	A*(h_1)	A*(h_2)	BFS	A*(h_1)	A*(h_2)
6	128	24	19	2,01	1,42	1,34
8	368	48	31	1,91	1,40	1,30
10	1033	116	48	1,85	1,43	1,27
12	2672	279	84	1,80	1,45	1,28
14	6783	678	174	1,77	1,47	1,31
16	17270	1683	364	1,74	1,48	1,32
18	41558	4102	751	1,72	1,49	1,34
20	91493	9905	1318	1,69	1,50	1,34
22	175921	22955	2548	1,66	1,50	1,34
24	290082	53039	5733	1,62	1,50	1,36
26	395355	110372	10080	1,58	1,50	1,35
28	463234	202565	22055	1,53	1,49	1,36

Można by zapytać, czy heurystyka h_2 jest zawsze lepsza od h_1?. Odpowiedź brzmi: „Zasadniczo tak" — a z definicji obu heurystyk wynika, że dla dowolnego węzła n zachodzi zależność $h_2(n) \geq h_1(n)$. Mówimy, że h_2 **dominuje** nad h_1. Dominacja ta przekłada się bezpośrednio na efektywność algorytmu: A* używający heurystyki h_2 nigdy nie rozwinie więcej węzłów niż A* używający h_1 (z wyjątkiem oczywiście remisu między h_1 a h_2). Uzasadnienie tego stwierdzenia jest proste. Jak wcześniej wyjaśnialiśmy, każdy węzeł, dla którego funkcja ewaluacyjna f zwraca wartość mniejszą niż C^*, zostanie niewątpliwie rozwinięty; jest to równoważne stwierdzeniu, że każdy węzeł n, dla którego $h(n) < C^* - g(n)$, zostanie niewątpliwie rozwinięty, o ile heurystyka h jest spójna. Ale ponieważ dla każdego węzła n zachodzi zależność $h_2(n) \geq h_1(n)$, każdy węzeł niewątpliwie rozwinięty przez wyszukiwanie (A*, h_2) zostałby niewątpliwie rozwinięty przez wyszukiwanie (A*, h_1), a ponadto (A*, h_1) może rozwijać jeszcze inne węzły. Generalnie więc korzystnie jest używać heurystyk zwracających większe wartości, pod warunkiem wszakże, iż heurystyki te są spójne, oraz ich obliczanie nie trwa zbyt długo w porównaniu z obliczaniem innych (stąd zastrzeżenie na wstępie, że są one „zasadniczo" lepsze).

3.6.2. Generowanie heurystyk na podstawie rozluźnionych problemów

Gdy analizuje się heurystyki h_1 (liczbę źle ulokowanych kafelków) i h_2 (odległość Manhattanu) dla problemu układanki puzzle-8 oraz przewagę drugiej z nich nad pierwszą, można się zastanawiać, skąd biorą się pomysły na tego typu heurystyki i — co ważniejsze — czy komputery są w stanie mechanicznie konstruować takowe?

Wartością $h_1(n)$ i $h_2(n)$ jest oszacowanie długości ścieżki na odcinku od węzła n do węzła docelowego; są to heurystyki adekwatne dla oryginalnego problemu puzzle-8, lecz także dla jego wersji rozszerzonej. Wyobraźmy sobie mianowicie, iż reguły gry rozluźnione zostały w ten sposób, iż kafelek może przesuwać się dokądkolwiek, nie tylko na sąsiednie puste pole; wtedy h_1 nadal powinna zwracać dokładną długość najkrótszego rozwiązania. Podobnie, jeśli dopuścimy, by kafelek mógł przesuwać się na sąsiednie pole w dowolnym kierunku, nawet na pole zajęte już przez inny kafelek, heurystyka h_2 nadal powinna zwracać dokładną długość najkrótszego rozwiązania. Ogólnie, problem będący wynikiem uproszczenia oryginalnego problemu, w sensie zmniejszenia ograniczeń nałożonych na możliwe akcje, nosi nawę **problemu rozluźnionego** (ang. *relaxed problem)*. Graf odzwierciedlający przestrzeń stanów problemu rozluźnionego jest *supergrafem* w stosunku do grafu przestrzeni stanów problemu oryginalnego, ponieważ zawiera w stosunku do niego dodatkowe krawędzie będące efektem zniesienia poszczególnych ograniczeń.

Ponieważ graf problemu rozluźnionego różni się od grafu problemu oryginalnego tylko dodatkowymi krawędziami, przeto optymalne rozwiązanie oryginalnego problemu jest — z definicji — rozwiązaniem także problemu rozluźnionego, jednakże problem rozluźniony może mieć lepsze rozwiązanie, jeśli dodatkowe krawędzie jego grafu wnoszą do wynikowej ścieżki jakieś skróty. Zatem *koszt optymalnego rozwiązania problemu rozluźnionego stanowi dopuszczalną heurystykę dla problemu oryginalnego*. Co więcej, ponieważ otrzymana w ten sposób heurystyka odzwierciedla dokładny koszt rozluźnionego problemu, musi ona spełniać nierówność trójkąta, jest więc heurystyką dopuszczalną.

Możliwości rozluźniania oryginalnego problemu stają się bardziej widoczne, jeśli zapiszemy ów problem w jakimś języku programowania[20]. Dla oryginalnej postaci puzzle-8 pojedynczy ruch kafelka można zapisać mniej więcej w taki sposób:

```
Kafelek można przesunąć z kwadratu X na kwadrat Y pod warunkiem, że
  (kwadraty X i Y sąsiadują ze sobą) i (kwadrat Y jest pusty)
```

Eliminując selektywnie jeden z warunków ujętych w nawiasy lub oba warunki, możemy uzyskać trzy rozluźnione problemy:

(a)

```
Kafelek można przesunąć z kwadratu X na kwadrat Y pod warunkiem, że
  (kwadraty X i Y sąsiadują ze sobą)
```

(b)

```
Kafelek można przesunąć z kwadratu X na kwadrat Y pod warunkiem, że
  (kwadrat Y jest pusty)
```

(c)

```
Kafelek można przesunąć z kwadratu X na kwadrat Y
```

Problem (a) jest źródłem heurystyki h_2 (odległości Manhattanu) zgodnie z wymaganiem, że heurystyka ta powinna pozostawać adekwatna, gdy przesuniemy jakiś kafelek w kierunku jego docelowej lokalizacji. Heurystyka wyprowadzona z problemu (b) jest przedmiotem ćwiczenia online 3.GASC. Z problemu (c) można wyprowadzić heurystykę h_1 (której wartością jest liczba niewłaściwie ulokowanych kafelków) — powinna ona pozostawać adekwatna w sytuacji, jeśli kafelki będą mogły przemieszczać się do swych docelowych kwadratów w jednym ruchu (czyli w pojedynczej akcji).

[20] W rozdziałach 8. i 11. opisujemy formalne języki przydatne do takiego zadania — zapis problemu w takim języku umożliwia mechaniczne wyprowadzanie z niego problemów rozluźnionych. Obecnie wystarczy nam zapis w języku potocznym.

Zauważmy, że rozluźnione problemy generowane w opisany sposób mogą być rozwiązywane *bez wyszukiwania*, ponieważ rozluźnienie reguł pozwala na dekompozycję problemu na osiem niezależnych podproblemów. Jeśli rozluźniony problem jest trudno rozwiązywalny, obliczanie wartości wyprowadzonej z niego heurystyki jest kosztowne.

Możliwe jest automatyczne generowanie heurystyk na podstawie definicji problemu i wywodzących się z niego problemów rozluźnionych, czego dowodem jest program ABSOLVER, wykorzystujący jeszcze inne techniki pomocnicze (Prieditis, 1993). Zespół twórców tego programu wygenerował za jego pomocą nową heurystykę dla układanki puzzle-8, lepszą niż wszystkie dotąd znane; znalazł także pierwszą użyteczną heurystykę dla kostki Rubika.

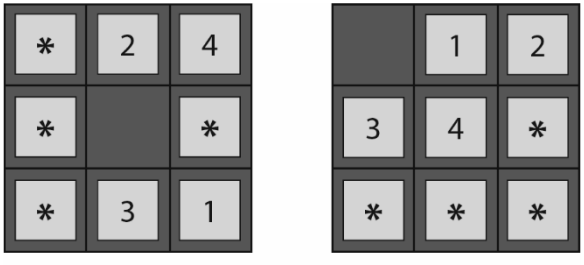
Jeśli dla danego problemu dostępny jest zbiór heurystyk dopuszczalnych h_1 ... h_m i żadna z nich nie jest wyraźnie lepsza od pozostałych, to którą należałoby wybrać? Okazuje się, że wcale nie musimy wybierać — możemy spożytkować zalety wszystkich, definiując nową:

$$h(n) = max\big(h_1(n),\ \cdots, h_m(n)\big)$$

Ta złożona heurystyka wybiera funkcję, która jest najbardziej adekwatna dla danego węzła. Ponieważ składowe heurystyki h_i są heurystykami dopuszczalnymi, h również jest takową; jeśli wszystkie są spójne, h jest również spójna. Z definicji h dominuje nad wszystkimi swymi składowymi. Mankamentem takiej złożonej heurystyki jest dłuższy czas jej obliczania, wynikający (zazwyczaj) z konieczności obliczenia wszystkich heurystyk składowych; jeśli staje się on problemem, można go zniwelować, wybierając do obliczenia *tylko jedną* z heurystyk składowych — losowo lub (lepiej) na podstawie algorytmu uczenia maszynowego przewidującego, która będzie najlepsza dla danego węzła. Może to skutkować utratą spójności wynikowej heurystyki, ale zwykle prowadzi do szybszego rozwiązywania problemów.

3.6.3. Generowanie heurystyk na podstawie podproblemów — bazy wzorców

Heurystyki (dopuszczalne) dla danego problemu można także wyprowadzać na podstawie kosztów rozwiązań jego **podproblemów**. Na rysunku 3.19 widzimy instancję podproblemu układanki puzzle-8 z rysunku 3.18, polegającego na umieszczeniu we właściwych miejscach kafelków 1, 2, 3, 4 i pustego kwadratu. Rzecz jasna, koszt optymalnego rozwiązania tego podproblemu wyznacza dolne ograniczenie dla kosztu optymalnego rozwiązania pełnego problemu; w niektórych przypadkach okazuje się to oszacowaniem dokładniejszym niż heurystyka odległości Manhattanu.



RYSUNEK 3.19. Instancja podproblemu układanki puzzle-8 z rysunku 3.18 polegającego na sprowadzeniu na docelowe lokalizacje kafelków 1, 2, 3, 4i pustego miejsca, bez zajmowania się innymi kafelkami

Pomysł **baz wzorców** (ang. *pattern databases*) polega na przechowywaniu dokładnych kosztów rozwiązania dla każdej możliwej instancji podproblemu — w naszym przykładzie, każdej możliwej konfiguracji wymienionych czterech kafelków i pustego miejsca (w bazie będzie wówczas 9×8×7×6×5 = 15 120 wzorców; położenia pozostałych czterech kafelków są nieistotne z perspektywy rozwiązywania tego konkretnego podproblemu, ale ich ruchy mogą mieć wpływ na koszt jego rozwiązania). Następnie obliczamy heurystykę h_{DB} dla każdego stanu występującego w wyszukiwaniu, po prostu odczytując jej wartość z bazy dla określonej konfiguracji. Sama baza konstruowana jest poprzez wyszukiwanie wstecz począwszy od stanu docelowego i rejestrowanie kosztów każdego nowo napotkanego wzorca[21]. Koszty takiego wyszukiwania amortyzują się w kolejnych instancjach problemu, ma więc ono sens, jeśli raz zbudowana baza wzorców będzie często wykorzystywana do rozwiązywania wielu problemów.

Wybór kafelków 1-2-3-4 jest całkowicie dowolny, równie dobrze moglibyśmy wybrać kafelki 5-6-7-8 lub 2-4-6-8. Każda baza wzorców dostarcza pewnej heurystyki, a poszczególne heurystyki można ze sobą łączyć (wybierając maksymalną, jak wcześniej wyjaśnialiśmy). Skonstruowana w ten sposób złożona heurystyka jest dokładniejsza niż heurystyka odległości Manhattanu — w przypadku odmiany układanki z 15 kafelkami na planszy 4×4 liczba generowanych węzłów zmniejsza się 1000-krotnie. Każda nowa baza wzorców wiąże się jednak z dodatkowymi wymogami pamięciowymi i dodatkowym czasem obliczeń, co zmniejsza zysk netto kosztu rozwiązania problemu.

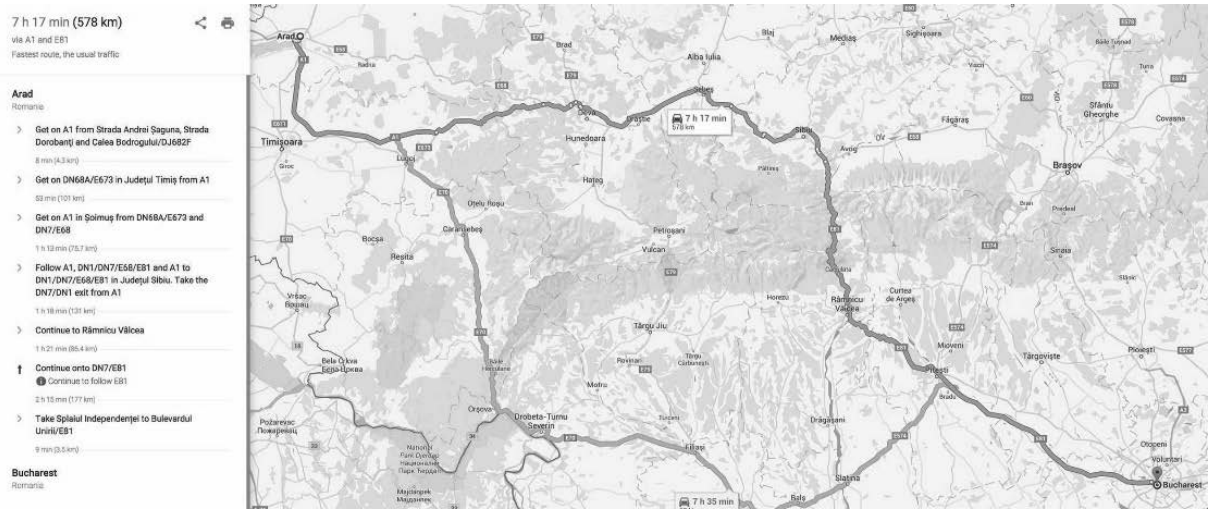
Można się zastanawiać, czy heurystyki uzyskane z baz wzorców dla konfiguracji 1-2-3-4 i 5- 6-7-8 można do siebie dodawać, skoro te dwa podproblemy nie pokrywają się — czy wynikowa heurystyka byłaby dopuszczalna? Odpowiedź brzmi „nie", ponieważ oba wymienione podproblemy prawie na pewno będą miały „część wspólną", czyli wspólne ruchy dla danego stanu — jest wręcz niemożliwe przesunięcie kafelków 1-2-3-4 na ich miejsca docelowe bez ruszania kafelków 5-6-7-8, i vice versa. Ale co się stanie, jeśli nie uwzględnimy ruchów kafelków 5-6-7-8 — jeżeli, zamiast abstrahować te kafelki do postaci gwiazdek, sprawimy, że po prostu znikną z planszy? Będzie to oznaczać, że nie rejestrujemy całkowitego rozwiązania podproblemu 1-2-3-4, ale jedynie zliczamy ruchy tych właśnie kafelków. Wtedy obie bazy wzorców będą **rozłączne**, suma obu kosztów — liczby ruchów kafelków 1-2-3-4 i kosztu rozwiązania podproblemu 5-6-7-8 — nadal będzie dolnym ograniczeniem kosztu rozwiązania całego problemu. Dzięki takiej rozłączności rozwiązanie problemu układanki puzzle-15 skróci się 10 000 razy w porównaniu ze stosowaniem heurystyki odległości Manhattanu, i trwać będzie zaledwie kilka milisekund. W przypadku 24 kafelków na planszy 5×5 rozwiązywanie problemu skróci się milion razy. Rozłączne bazy wzorców sprawdzają się w przypadku problemu przesuwanych kafelków, ponieważ problem ten daje się podzielić na podproblemy w taki sposób, że jedna akcja związana jest z dokładnie jednym podproblemem (w jednym ruchu przesuwamy dokładnie jeden kafelek).

3.6.4. Generowanie heurystyk w oparciu o punkty orientacyjne

Istnieją usługi online, które udostępniają mapy z dziesiątkami milionów lokalizacji i w ciągu milisekund znajdują optymalne kosztowo trasy przejazdu między nimi (rysunek 3.20). Jak to jest możliwe, wszak algorytmy wyszukiwania, które opisywaliśmy do tej pory, są miliony razy wolniejsze? Jest to wynik wielu trików, ale podstawowa tajemnica kryje się w **preobliczeniach** (ang. *precomputations*), czyli wstępnych obliczeniach optymalnych kosztów ścieżek. Są to obliczenia niewątpliwie czasochłonne, ale wykonywane są tylko raz i czasochłonność ta amortyzuje się niemal natychmiast wobec miliardów żądań napływających od użytkowników.

Moglibyśmy wygenerować doskonałą heurystykę przez wstępne obliczenie i przechowywanie kosztu optymalnej ścieżki między każdą parą wierzchołków. Jeśli potraktujemy mapę jako graf, którego wierzchołki V są lokalizacjami, a krawędzie E drogami łączącymi te lokalizacje, to obliczenie takiej heurystyki wymagałoby $O(|V|^2)$ pamięci i $O(|E|^3)$ czasu — rzecz wykonalna w praktyce dla grafu liczącego 10 000 wierzchołków, ale nie 10 milionów!

[21] Dzięki wyszukiwaniu wstecz począwszy od celu otrzymujemy natychmiast dokładny koszt rozwiązania każdej napotkanej instancji. Jest to przykład **programowania dynamicznego**, którym zajmiemy się dokładniej w rozdziale 17.



RYSUNEK 3.20. Usługa webowa umożliwiająca wytyczanie tras, funkcjonująca w oparciu o algorytmy wyszukiwania

Podejście bardziej praktyczne polega więc na wybraniu kilku (być może 10 lub 20) **punktów orientacyjnych** (ang. *landmarks*)[22] spośród lokalizacji (wierzchołków), po czym dla każdego punktu orientacyjnego L należy obliczyć dokładne koszty optymalnych ścieżek $C^*(v, L)$ łączących go z każdym z pozostałych wierzchołków v (będziemy także potrzebować wartości $C^*(L, v)$ — w grafie nieskierowanym wartości te byłyby identyczne, ale w grafie skierowanym, koniecznym choćby dla uwzględnienia dróg jednokierunkowych, należy obliczać je oddzielnie. Dysponując tablicą wartości C* możemy łatwo skonstruować następującą efektywną (choć niekoniecznie dopuszczalną) heurystykę dla węzła n: dla wszystkich punktów orientacyjnych L obliczamy optymalny koszt ścieżki prowadzącej od węzła n do celu poprzez L i wybieramy najmniejszą z tych wartości:

$$h_l(n) = \min_{L \in \text{punkty orientacyjne}} C^*(n, L) + C^*(L, cel)$$

Jeśli optymalna ścieżka przebiegać będzie przez punkt orientacyjny, heurystyka ta będzie dokładna; jeśli nie, heurystyka ta nie będzie dopuszczalna, bo koszt celu będzie zawyżony. Jeśli w wyszukiwaniu A* używamy dokładnej heurystyki, to za każdym razem, gdy osiągniemy węzeł nieleżący na optymalnej ścieżce, każdy węzeł wygenerowany wskutek jego rozwinięcia będzie leżał na optymalnej ścieżce. Wyobraźmy sobie kontury tworzące się wokół tej optymalnej ścieżki: wyszukiwanie będzie nią podążać, dodając w każdej iteracji akcje o koszcie c i przechodząc tym samym do nowego stanu, dla którego wartość h będzie mniejsza o c od tej dla stanu bieżącego. Oznacza to, że wartość funkcji $f = g+h$ powinna wynosić niezmiennie C^* wzdłuż całej ścieżki.

Niektóre algorytmy znajdowania tras wykonują się jeszcze szybciej, dzięki **skrótom** (ang. *shortcuts*) — dodanym do grafu sztucznym krawędziom, z których każda reprezentuje optymalną ścieżkę złożoną z wielu akcji. Wyobraźmy sobie, że dysponujemy takimi skrótami między każdą parą 100 największych miast USA; jeśli chcemy dojechać z kampusu Uniwersytetu Berkeley w Kalifornii do gmachu Uniwersytetu w Nowym Jorku, to możemy wykorzystać skrót między Sacramento a Manhattanem, przemierzając 90% trasy w ramach jednej akcji.

[22] Zwanych niekiedy także „centrami" (ang. *pivots*) lub „kotwicami" (ang. *anchors*).

Heurystyka $h_L(n)$ jest efektywna, ale nie jest dopuszczalna. Po krótkim zastanowieniu się możemy jednak sformułować podobną heurystykę, która jest zarówno efektywna, jak i dopuszczalna.

$$h_{DH}(n) = \max_{L \in \text{punkty orientacyjne}} |C^*(n, L) - C^*(cel, L)|$$

Heurystyka na nosi nazwę **różnicowej** (ang. *differential heuristic*) ze względu na występujące w niej odejmowanie. Jeżeli *cel* znajduje się na optymalnej ścieżce od *n* do *L*, to widać wyraźnie, że odległość od *n* do celu szacowana jest przez h_{DH} jako różnica kosztu odcinków (*n*, *L*) i (*cel*, *L*). Jeżeli cel leży nieco poza optymalną ścieżką od *n* do *L*, h_{DH} nie jest już heurystyką dokładną, ale wciąż jest dopuszczalna. Punkt orientacyjny, który znajduje się na optymalnej ścieżce od *n* do *celu*, jest raczej bezużyteczny; gdy znajduje się on w połowie drogi między *n* a *celem*, wartością $h_{DH}(n)$ jest zero.

Punkty orientacyjne można wybierać na wiele sposobów. Najprostszy z nich, wybór losowy, jest szybki, ale mało użyteczny, gdy wiele punktów orientacyjnych znajdzie się zbyt bliskie siebie. Można minimalizować prawdopodobieństwo takich sytuacji, stosując *zachłanną* strategię wyboru: pierwszy punkt orientacyjny wybieramy losowo, a następnie w każdej z kolejnych iteracji ustalamy punkt orientacyjny, który znajduje się jak najdalej od najbliższego z istniejących już punktów orientacyjnych (przeczytaj to jeszcze raz uważnie). Najbardziej elastyczną strategią jest dobór — i okresowe jego aktualizowanie — punktów orientacyjnych na podstawie historii żądań użytkowników. Z perspektywy heurystyki różnicowej najkorzystniejszym układem są punkty orientacyjne równomiernie rozłożone blisko krawędzi mapy; zbudowanie takiego układu wymaga wykonania następujących kroków:

1. Wyliczamy współrzędne (*cx*, *cy*) „środka ciężkości" (centroidu) wszystkich lokalizacji na mapie

$$cx = \frac{1}{N}\sum_{i=1}^{N} x_i$$

$$cy = \frac{1}{N}\sum_{i=1}^{N} y_i$$

   gdzie (x_i, y_i), i = 1, ..., *N* są współrzędnymi lokalizacji.

2. Ustalamy liczbę punktów orientacyjnych (*K*) i z punktu (*cx*, *cy*) wyprowadzamy *K* półprostych, w miarę równomiernie rozłożonych kątowo.
3. Jako punkty orientacyjne wybieramy lokalizacje położone jak najbliżej tych półprostych i jednocześnie jak najdalej od centroidu.

Strategia punktów orientacyjnych sprawdza się bardzo dobrze w problemach wyszukiwania tras również ze względu na projektowanie układu dróg na świecie: ponieważ większość użytkowników preferuje wybór trasy właśnie w oparciu o punkty orientacyjne, inżynierowie dążą do projektowania najszerszych i najszybszych tras właśnie między tymi punktami. Wyszukiwanie wykorzystujące heurystyki bazujące na punktach orientacyjnych preferuje wybór takich właśnie tras.

3.6.5. Uczenie maszynowe dla lepszego wyszukiwania

Przedstawiane dotychczas strategie wyszukiwania — wyszukiwanie wszerz, wyszukiwanie A* itd. — to ustalone algorytmy starannie zaprojektowane i zaprogramowane przez informatyków. Powstaje w tym momencie pytanie, czy agent potrafiłby *nauczyć się* lepszego wyszukiwania? Odpowiedź jest twierdząca, a sposób, w jaki agent mógłby się uczyć, opiera się na ważnej koncepcji **metapoziomowej przestrzeni stanów** (ang. *metalevel state space*). Każdy stan w tej metapoziomowej przestrzeni reprezentuje wewnętrzny (obliczeniowy) stan programu przeszukującego „zwykłą" przestrzeń stanów, taką jak na przykład mapa Rumunii. Aby odróżnić te dwa typy przestrzeni, będziemy tę „zwykłą"

nazywać **obiektową przestrzenią stanów** (ang. *object-level state space*). Na przykład aktualny stan wewnętrzny algorytmu A* można utożsamiać z bieżącą postacią drzewa wyszukiwawczego; każda akcja w tej metapoziomowej przestrzeni jest krokiem algorytmu zmieniającym stan wewnętrzny — w algorytmie A* taki krok polega na rozwinięciu liścia i dodaniu do drzewa wygenerowanych węzłów potomnych. Rysunek 3.12, przedstawiający kolejne stadia rozwoju drzewa wyszukiwawczego, może być traktowany jako ilustracja *budowania ścieżki w przestrzeni metapoziomowej* — każdy stan na tej ścieżce odpowiada konkretnej postaci drzewa wyszukiwawczego w przestrzeni obiektowej.

Wspomniana „ścieżka" z rysunku 3.12 składa się z pięciu etapów, z których jeden — rozwinięcie przez Fagaras — nie jest szczególnie użyteczny. W przypadku trudniejszych problemów wyszukiwania zdarza się wiele takich chybionych etapów, a agent, wykorzystując algorytm **uczenia metapoziomowego** (ang. *meta-level learning*), zyskuje pole do popisu pod względem umiejętności unikania takich etapów, czyli rezygnacji z eksplorowania mało obiecujących poddrzew. Techniki edukowania agenta w tym kierunku zostały opisane w rozdziale 22., a ich zadaniem jest minimalizowanie **całkowitego kosztu** rozwiązywania problemów, droga kompromisów między kosztem ścieżki a złożonością obliczeniową.

3.6.6. Wynajdywanie heurystyk przez doświadczenie

Opisaliśmy wcześniej jeden ze sposobów konstruowania użytecznych heurystyk — rozwiązywanie „rozluźnionych" wersji oryginalnego problemu, dla których można łatwo znajdować optymalne rozwiązania. Alternatywnym sposobem jest wynajdywanie heurystyk drogą uczenia się przez doświadczenia — zilustrujemy tę koncepcję na przykładzie układanki puzzle-8. Zdobywanie „doświadczenia" polega na rozwiązywaniu wielu takich układanek. Każde optymalne rozwiązanie układanki jest przykładową parą (cel, ścieżka); algorytm uczący może zostać wykorzystany do skonstruowania, na podstawie kolekcji takich par, funkcji heurystycznej h, od której można oczekiwać przybliżania prawdziwego kosztu ścieżki dla innych stanów, które mogą się pojawiać podczas wyszukiwania. Wiele podejść do tego tematu skutkuje niedoskonałą aproksymacją, wskutek czego otrzymana heurystyka może nie być dopuszczalna. Prowadzi to do konieczności rozstrzygania nieuniknionych kompromisów między czasem uczenia się, czasem wyszukiwania i kosztem otrzymywanych rozwiązań. Techniki uczenia maszynowego prezentujemy w rozdziale 19. Uczenie ze wzmacnianiem, opisywane w rozdziale 22., również ma zastosowanie w wyszukiwaniu.

Niektóre techniki uczenia maszynowego spisują się lepiej, gdy opisowi każdego stanu towarzyszy zestaw **cech** (ang. *features*) pomocnych w przewidywaniu wartości heurystyki dla tego stanu, na przykład cecha „liczba źle ulokowanych kafelków" może być pomocna w szacowaniu odległości aktualnego stanu układanki od stanu docelowego; oznaczmy tę cechę przez $x_1(n)$. Moglibyśmy wygenerować losowo 100 różnych konfiguracji układanki i zebrać statystykę rzeczywistych kosztów ich rozwiązań. I może na przykład okazać się, że przy wartości $x_1(n) = 5$ średni koszt rozwiązania wynosi około 14, itp.; oczywiście możemy w opisach stanów wykorzystywać wiele cech — drugą pomocną cechą stanu może być „liczba par kafelków, które sąsiadują ze sobą w bieżącym stanie i nie sąsiadują w stanie docelowym"; oznaczmy tę cechę przez $x_2(n)$. Jest zrozumiałe, że w celu otrzymania użytecznej heurystyki powinniśmy *jakoś* połączyć obie te cechy. Najczęściej pod słowem „jakoś" kryje się kombinacja liniowa cech:

$$h(n) = c_1x_1(n)+c_2x_2(n)$$

Wartości stałych c_1 i c_2 należy dobrać eksperymentalnie, drogą najlepszego dostosowania funkcji h do danych eksperymentalnych otrzymanych w losowo wygenerowanych konfiguracjach. Można się spodziewać, że c_1 i c_2 będą dodatnie, z prostego względu — większe wartości cech x_1 i x_2 czynią problem trudniejszym do rozwiązania. Zauważmy, że heurystyka h zwraca wartość 0 dla węzłów docelowych, ale niekoniecznie musi być dopuszczalna ani spójna.

Podsumowanie

W tym rozdziale przedstawiliśmy algorytmy wyszukiwania, które agent stosować może do wybierania sekwencji działań w wielu różnych środowiskach — pod warunkiem, że są to środowiska epizodyczne, jednoagentowe, w pełni obserwowalne, deterministyczne, statyczne, dyskretne i rozpoznane. Wyszukiwanie wiąże się nieuchronnie ze znajdowaniem kompromisów między czasem wykonywania algorytmów, ich zapotrzebowaniem na pamięć i jakością zwracanych rozwiązań. Możemy zwiększyć efektywność wyszukiwania, jeżeli wykorzystamy wiedzę zależną od domeny problemu i zakodujemy ją w formie funkcji heurystycznej, której wartość dla danego stanu jest oszacowaniem odległości tego stanu od stanu docelowego. Innym sposobem na przyspieszenie wyszukiwania jest dokonanie pewnych obliczeń wstępnych (związanych z punktami orientacyjnymi) i wykorzystywanie ich wyników.

- Wyszukiwanie można rozpocząć dopiero po ścisłym sformułowaniu definicji **problemu**.
- Definicja problemu zawiera pięć elementów: **stan początkowy**, zbiór **akcji**, **model przejścia** (opisujący efekty zastosowania konkretnych akcji do konkretnych stanów), zbiór **stanów docelowych** i funkcję określającą **koszt poszczególnych akcji**.
- Środowisko problemu reprezentowane jest przez **graf przestrzeni stanów**. **Ścieżka** (ciąg akcji) w przestrzeni stanów, od stanu początkowego do stanu docelowego, stanowi **rozwiązanie** problemu.
- Z perspektywy algorytmów wyszukiwania stany i akcje są (generalnie) **niepodzielne** — nieistotna jest ich wewnętrzna struktura — chociaż można przypisywać poszczególnym stanom cechy, przydatne w procesie uczenia się agenta.
- Podstawowymi kryteriami oceny algorytmów wyszukiwania są: **zupełność**, **optymalność kosztowa**, **złożoność czasowa** i **złożoność pamięciowa**.
- **Wyszukiwanie niedoinformowane** to algorytm, dla którego jedynym źródłem wiedzy jest definicja rozwiązywanego problemu. Działanie algorytmu wyszukiwania opiera się na budowaniu, krok po kroku, drzewa wyszukiwawczego. Ze względu na strategię owego budowania — czyli kryterium wyboru węzłów do rozwijania — rozróżniany następujące metody wyszukiwania:
  - **„najpierw najlepszy"** — do rozwinięcia wybierany jest węzeł, dla którego funkcja ewaluacyjna zwraca najmniejszą wartość;
  - **wyszukiwanie wszerz** rozwija w pierwszej kolejności węzły o najmniejszej głębokości; wyszukiwanie to jest zupełne, optymalne pod względem liczby akcji w rozwiązaniu, lecz posiada wykładniczą złożoność pamięciową;
  - **wyszukiwanie przy jednolitym koszcie** rozwija w pierwszej kolejności węzły o najmniejszym koszcie ścieżki ($g(n)$), jest optymalne pod względem całkowitego kosztu rozwiązania;
  - **wyszukiwanie w głąb** wybiera do rozwinięcia węzły położone najgłębiej w drzewie (te, które jeszcze nie zostały rozwinięte); nie jest ani zupełne, ani optymalne kosztowo, lecz jego zaletą jest liniowa złożoność pamięciowa. **Ograniczone wyszukiwanie w głąb** ignoruje węzły, których głębokość w drzewie przekracza pewien ustalony limit;
  - **wyszukiwanie z iteracyjnym zagłębianiem** to iteracyjne powtarzanie ograniczonego wyszukiwania w głąb, z sukcesywnie zwiększanym, w każdej iteracji, ograniczeniem głębokości — aż do osiągnięcia węzła docelowego. Jest algorytmem zupełnym, a jeśli przeprowadzana jest kompletna eliminacja cykli, jest optymalny pod względem liczby akcji w rozwiązaniu. Jego złożoność czasowa jest podobna jak przy wyszukiwaniu wszerz, natomiast jego złożoność pamięciowa jest liniowa.
  - **wyszukiwanie dwukierunkowe** działa w dwóch przeciwległych kierunkach — od węzła początkowego i od węzła docelowego — budując dwie niezależne granice, których spotkanie się kończy wykonywanie algorytmu.

- Metody **wyszukiwania poinformowanego** wykorzystują **funkcje heurystyczne**; wartość takiej funkcji $h(n)$ jest oszacowaniem częściowego kosztu rozwiązania, od węzła n do węzła docelowego. Metody te mogą również wykorzystywać inne źródła informacji, na przykład bazy wzorców zawierające koszty rozwiązań wybranych problemów. Ważniejsze z metod tej kategorii to:
  - **zachłanna wersja wyszukiwania „najpierw najlepszy"** — wybiera do rozwijania węzły z najmniejszą wartością h. Nie jest optymalna kosztowo, lecz zwykle jest efektywna;
  - **wyszukiwanie A*** rozwija w pierwszej kolejności węzły z najmniejszą wartością funkcji ewaluacyjnej $f(n) = g(n)+h(n)$. Jest algorytmem zupełnym i optymalnym kosztowo, pod warunkiem, że heurystyka h jest heurystyką dopuszczalną. Duża złożoność pamięciowa tej metody wciąż stanowi poważne wyzwanie w przypadku większości problemów;
  - **dwukierunkowe wyszukiwanie A*** jest niekiedy bardziej efektywne niż wersja jednokierunkowa;
  - **A* z iteracyjnym zagłębianiem (IDA*)** to połączenie klasycznego wyszukiwania A* z iteracyjnym zagłębianiem, cechuje się mniejszymi wymaganiami pamięciowymi niż klasyczne A*;
  - **rekurencyjne wyszukiwanie „najpierw najlepszy" (RBFS)** i **A* ograniczone pamięciowo (SMA*)** to solidne, optymalne kosztowo algorytmy zdolne do efektywnego wykorzystywania pamięci w granicach jej dostępności: przydatne do rozwiązywania problemów, dla których wymagania pamięciowe klasycznego A* są dyskwalifikujące;
  - **wyszukiwanie skupione (wiązkowe)** narzuca górny limit na liczbę węzłów tworzących granicę. Nie jest algorytmem zupełnym ani optymalnym kosztowo, zwracane rozwiązania są jednak zwykle bliskie optymalności, poza tym czas wykonania jest znacząco krótszy w porównaniu z algorytmami zupełnymi;
  - **ważone wyszukiwanie A*** koncentruje się na szybkim zmierzaniu do celu, nie gwarantując optymalności kosztowej.

Wydajność algorytmów wyszukiwania wykorzystujących heurystykę zależy przede wszystkim od jakości funkcji heurystycznej. Źródłami dobrych heurystyk mogą być: optymalne rozwiązania rozluźnionych wersji oryginalnego problemu, obliczone wcześniej i zapamiętane w bazach wzorców wartości kosztów rozwiązań podproblemów, oraz uczenie maszynowe przez doświadczenie w rozwiązywaniu problemów danej klasy.

Bibliografia i uwagi historyczne

Temat wyszukiwania w przestrzeni stanów pojawił się w historii sztucznej inteligencji dość wcześnie. Prace Newella i Simona dotyczące teorii logiki (1957) i GPS (1961) doprowadziły do ustanowienia algorytmów wyszukiwania jako podstawowego narzędzia dla badaczy z lat 60. ubiegłego wieku, a rozwiązywanie problemów uznane zostało za kanoniczne zadanie sztucznej inteligencji. Praca Richarda Bellmana (1957) z dziedziny badań operacyjnych wykazała znaczenie addytywności kosztów ścieżek w upraszczaniu algorytmów optymalizacji. Tekst Nilsa Nilssona (1971) dostarczył temu obszarowi solidnych podstaw teoretycznych.

Układanka puzzle-8 jest „uboższym kuzynem" układanki puzzle-15, której historię szczegółowo opisują Slocum i Sonneveld (2006). W 1880 roku puzzle-15 zwróciły uwagę opinii publicznej i matematyków (Johnson i Story, 1879; Tait, 1880) — redaktorzy „American Journal of Mathematics" stwierdzili: „układanka puzzle-15 w ciągu ostatnich kilku tygodni stała się tematem nr 1 w opinii publicznej, zyskując sobie uznanie dziewięciu na dziesięciu Amerykanów, niezależni od płci, wieku i pozycji społecznej". 12 marca 1880 roku, na łamach tygodnika „Weekly News-Democrat" w Emporia (w stanie Kansas), porównano wręcz tę układankę do „epidemii, która ogarnęła cały kraj".

Sam Loyd, słynny amerykański projektant gier, ogłosił się samozwańczo wynalazcą puzzle-15 (Loyd, 1959), faktycznie jednak wymyślił ją w latach 70. XIX wieku Nouyes Chapman, listonosz z wioski Canastota w stanie Nowy Jork; oficjalny patent na łamigłówkę z przesuwanymi kafelkami przyznano jednak w 1878 roku Ernestowi Kinseyowi. Ratner i Warmuth (1986) wykazali, że ogólna wersja puzzli na planszy $n \times n$ (puzzle-k, gdzie $k = n^2-1$) należy do klasy problemów NP-zupełnych (patrz Dodatek A).

Ernö Rubik wymyślił swą słynną kostkę w 1974 roku, odkrywając również algorytm jej dobrych, ale nie optymalnych, rozwiązań. Korf (1997) znalazł optymalne rozwiązania dla niektórych losowo wygenerowanych instancji problemu, używając wyszukiwania IDA* w połączeniu z bazami wzorców. Rokicki i in. (2014) udowodnili, że każdą konfigurację kostki Rubika można rozwiązać w maksymalnie 26 ruchach, jeśli obrót o 180 stopni potraktujemy jako dwa ruchy (w 20 ruchach, jeśli potraktujemy ów obrót jako jeden ruch); znalezienie tego dowodu pochłonęło 35 lat (!) pracy procesora, lecz nie prowadziło bezpośrednio do wydajnego algorytmu. Agostinelli i in. (2019) wykorzystali uczenie maszynowe ze wzmacnianiem, głębokie uczenie i przeszukiwanie drzewa metodą Monte Carlo[23] do „wyuczenia" znacznie wydajniejszego *solvera*, dającego rozwiązanie w czasie krótszym niż sekunda, lecz bez gwarancji optymalności.

Każdy z przytaczanych w tym rozdziale przykładów wyszukiwania, zaczerpniętych z rzeczywistego świata, był przedmiotem wielu prac badawczych. Metody zestawiania optymalnych tras przelotów chronione są w większości tajemnicą i prawami autorskimi, niemniej jednak Carl de Marcken wykazał, że (wobec ogromnych komplikacji taryf i wielu innych ograniczeń) problem konstruowania optymalnego lotu jest formalnie nierozstrzygalny; swój wynik uzyskał przez redukcję oryginalnego problemu do (nierozstrzygalnych) diofantycznych problemów decyzyjnych (Robinson, 2002). Problem komiwojażera (TSP) jest jednym ze standardowych problemów kombinatorycznych w informatyce teoretycznej (Lawler i in., 1992). Karp (1972) udowodnił, że problem decyzyjny TSP[24] jest NP-trudny (patrz Dodatek A), ale opracowano skuteczne metody heurystycznej aproksymacji dokładnego rozwiązania (Lin i Kernighan, 1973). Arora (1998) opracował schemat (o złożoności wielomianowej) aproksymacji rozwiązania TSP w sytuacji, gdy koszty dróg są ich długościami w przestrzeni euklidesowej. Problem optymalnych topografii układów VLSI badany był przez LaPaugh (2010), a liczne artykuły poświęcone tej problematyce pojawiają się w specjalistycznych czasopismach. Nawigację robotów omawiamy w rozdziale 26. Automatyczne sekwencjonowanie montażu zostało po raz pierwszy zademonstrowane przez robota FREDDY (Michie, 1972), a obszernej recenzji dostarczają Bahubalendruni i Biswal (2016).

Niedoinformowane algorytmy wyszukiwania są jednym z głównych obszarów informatyki (Cormen i in., 2009) i badań operacyjnych (Dreyfus, 1969). Wyszukiwanie wszerz zostało sformułowane przez Moore'a (1959) w celu rozwiązywania labiryntów. Metoda programowania dynamicznego (Bellman, 1957; Bellman i Dreyfus, 1962), która systematycznie rejestruje rozwiązania dla wszystkich podproblemów o rosnącej długości, może być postrzegana jako odmiana wyszukiwania wszerz.

Algorytm Dijkstry — w formie, w jakiej jest zazwyczaj przedstawiany (Dijkstra, 1959) — ma zastosowanie do jawnych grafów skończonych. Nilsson (1971) przedstawił wersję algorytmu Dijkstry, którą nazwał „wyszukiwaniem przy jednolitym koszcie" (według jego sformułowania „rozciąga się wzdłuż konturów o jednakowych kosztach ścieżki"); wersja ta umożliwia wyszukiwane również w grafach niejawnie definiowanych i(lub) nieskończonych. W pracy Nilssona znajduje się wyjaśnienie koncepcji zamkniętych i otwartych list oraz „wyszukiwania grafowego". Nazwa `BEST-FIRST-SEARCH` pojawiła się po raz pierwszy w *Handbook of AI* (Barr i Feigenbaum, 1981). Algorytmy Floyda-Warshalla (Floyd, 1962) i Bellmana-Forda (Bellman, 1958; Ford, 1956) dopuszczają ujemne koszty akcji, o ile w grafie przestrzeni stanów nie występują cykle o ujemnym koszcie.

Pewna odmiana iteracyjnego zagłębiania, zaprojektowana w celu efektywnego wykorzystywania zegara szachowego, została po raz pierwszy zastosowana przez Slate'a i Atkina (1977) w programie CHESS 4.5. W algorytmie *B* Martellego (1977) również zauważyć można ślady takiego zagłębiania. Samo sformułowanie techniki iteracyjnego zagłębiania zawdzięczamy Bertramowi Raphaelowi (1976), a wyeksponowana ona została w pracy Korfa (1985a).

[23] Patrz *https://pl.wikipedia.org/wiki/Monte-Carlo_Tree_Search — przyp. tłum.*

[24] Czyli odpowiedź na pytanie o istnienie trasy komiwojażera o koszcie mniejszym niż ustalone K — *przyp. tłum.*

Wykorzystanie heurystyk w rozwiązywaniu problemów pojawia się we wczesnej pracy Simona i Newella (1958), ale sformułowanie „wyszukiwanie heurystyczne" i przykłady wykorzystywania funkcji heurystycznych szacujących odległość do celu, pojawiły się nieco później (Newell i Ernst, 1965; Lin, 1965). Zakrojone na szeroką skalę eksperymenty w zakresie wyszukiwania heurystycznego przeprowadzone zostały przez Dorana i Michiego (1966). Chociaż przeanalizowali długość ścieżki i jej „penetrancję" (czyli stosunek jej długości do całkowitej liczby odwiedzonych dotychczas węzłów), wydaje się, że zignorowali informacje dostarczone przez koszt ścieżki (funkcję $g(n)$). Algorytm A*, uwzględniający ten koszt w wyszukiwaniu heurystycznym, został opracowany przez Harta, Nilssona i Raphaela (1968). Dechter i Pearl (1985) badali warunki, pod jakimi A* jest optymalnie wydajny (pod względem liczby rozwiniętych węzłów).

W oryginalnej pracy poświęconej A* (Hart i in., 1968) sformułowany został warunek spójności dla funkcji heurystycznych. Warunek monotoniczności został wprowadzony przez Pohla (1977) jako prostszy zamiennik spójności, ale Pearl (1984) wykazał, że oba warunki są sobie równoważne.

Pohl (1977) był pionierem w badaniu związku między błędami funkcji heurystycznych a złożonością czasową wyszukiwania A* — podstawowe wyniki w tym zakresie uzyskano dla wyszukiwania drzewiastego z jednostkowymi kosztami akcji i zarówno jednym stanem docelowym (Pohl, 1977; Gaschnig, 1979; Huyn i in., 1980; Pearl, 1984), jak i wieloma stanami docelowymi (Dinh i in., 2007). Korf i Reid (1998) pokazali, jak można przewidzieć dokładną liczbę rozwiniętych węzłów, a nie tylko jej asymptotyczne przybliżenie, w różnych rzeczywistych domenach problemowych. Koncepcja „efektywnego czynnika rozgałęzienia" została zaproponowana przez Nilssona (1971) jako empiryczna miara wydajności algorytmu. Helmert i Röoger (2008) zauważyli, że przy wyszukiwaniu grafowym, w przypadku kilku dobrze znanych problemów, w których liczba węzłów na optymalnej ścieżce rośnie wykładniczo do rozmiaru problemu, wyszukiwanie A* wykazuje wykładniczą złożoność czasową.

Istnieje wiele odmian algorytmu A*. Pohl (1970) wprowadził wyszukiwanie ważone A*, a później jego wersję dynamiczną (1973), w której waga zmienia się wraz z głębokością drzewa. Ebendt i Drechsler (2009) dokonali syntezy znanych wyników i przeprowadzili analizę niektórych zastosowań. Hatem i Ruml (2014) zaprezentowali uproszczoną i ulepszoną wersję ważonego A*, łatwiejszą w implementacji. Wilt i Ruml (2014) zaproponowali alternatywę dla zachłannego wyszukiwania — szybkie wyszukiwanie koncentrujące się na minimalizacji czasu wyszukiwania, a w swej późniejszej pracy (2016) udowodnili, że najlepsze heurystyki dla wyszukiwania satysfakcjonującego różnią się od tych dla wyszukiwania optymalnego. Kilka przydatnych sztuczek ułatwiających implementowanie szybkiego wyszukiwania znajdą programiści w pracy Burnsa i in (2012), a Felner (2018) pokazuje jak zmienia się kod implementujący szybkie wyszukiwane w związku z wcześniejszym testowaniem węzła na okoliczność bycia węzłem docelowym.

Pomysłodawcą wyszukiwania dwukierunkowego jest Pohl (1971). Holte i in. (2016) opisują wersję wyszukiwania dwukierunkowego, która gwarantuje spotkanie granic pośrodku, dzięki czemu ma szersze zastosowanie. Eckerle i in. (2017) wprowadzają pojęcie węzła niewątpliwie rozwiniętego, opisują zbiór par takich węzłów i udowadniają, że żadne wyszukiwanie dwukierunkowe nie gwarantuje optymalnej wydajności. Algorytm NBS (Chen i in., 2017) jawnie wykorzystuje kolejkę par węzłów.

Kombinacja dwukierunkowego wyszukiwania A* i koncepcji punktów orientacyjnych stanowi podstawę usługi online umożliwiającej efektywne wyszukiwanie tras, udostępnianej przez Microsoft (Goldberg i in., 2006). Po wstępnym obliczeniu i zapamiętaniu kosztów dróg łączących punkty orientacyjne, usługa potrafi znaleźć optymalną trasę między każdą parą spośród 24 milionów punktów na mapie USA, przeszukując zaledwie 0,1% ich populacji. Korf (1987) demonstruje kilka dodatkowych technik — podcele, makrooperatory i abstrakcje — które pozwalają uzyskać jeszcze większe przyspieszenie podstawowych mechanizmów. Delling i in. (2009) pokazują, jak używać wyszukiwania dwukierunkowego, punktów orientacyjnych, struktury hierarchicznej i innych sztuczek do znajdowania optymalnych tras samochodowych. Anderson i in. (2008) opisują pokrewną metodę, zwaną **wyszukiwaniem precyzowanym** (ang. *coarse-to-fine search*), której istotą jest wykorzystywanie punktów orientacyjnych na różnych poziomach hierarchicznej abstrakcji. Korf (1987) opisuje warunki, przy spełnieniu których metoda ta zapewnia przyspieszenie w stopniu wykładniczym. Wyniki eksperymentów i analiz badających ilościowe aspekty wyszukiwania hierarchicznego prezentuje Knoblock w swej pracy (1991).

A* i inne algorytmy wyszukiwania w przestrzeniach stanów są ściśle związane z technikami **podziału i ograniczeń** (ang. *branch-and-bound*), stosowanymi powszechnie w badaniach operacyjnych (Lawler i Wood, 1966; Rayward-Smith i in., 1996). Kumar i Kanal (1988) podejmują próbę „wielkiej unifikacji" wyszukiwania heurystycznego, programowania dynamicznego oraz techniki podziału i ograniczeń, opatrując swą teorię akronimem CDP, od ang. *composite decision process* — „złożony proces decyzyjny".

Ponieważ typowy komputer w latach 60. ubiegłego wieku posiadał co najwyżej kilka tysięcy słów pamięci, obiektem intensywnych badań stały się wówczas metody wyszukiwania z ograniczeniami pamięciowymi. Program *Graph Traverser* (Doran i Michie, 1966), jeden z pierwszych tego rodzaju, wybiera najlepszą akcję znalezioną przez wyszukiwanie „najpierw najlepszy" prowadzone aż do wyczerpania dostępnej pamięci. Pierwszym powszechnie stosowanym, optymalnym pod względem długości ścieżki wynikowej, heurystycznym algorytmem wyszukiwania z ograniczeniami pamięciowymi był IDA* (Korf, 1985b), nic więc dziwnego w tym, że opracowano sporo jego wariantów. Analiza efektywności algorytmu IDA* i jego trudności związanych z wykorzystywaniem *funkcji heurystycznych zwracających liczby rzeczywiste* pojawia się w pracy Patrick i in. (1992).

Oryginalna wersja algorytmu RBFS (Korf, 1993) jest faktycznie nieco bardziej skomplikowana niż algorytm pokazany na listingu 3.5, któremu w rzeczywistości bliżej jest do (opracowanego niezależnie) algorytmu zwanego **iteracyjnym rozwijaniem** (ang. IE — *iterative expansion*) (Russell, 1992). RBFS używa zarówno dolnego, jak i górnego ograniczenia, oba algorytmy zachowują się identycznie w przypadku heurystyki dopuszczalnej, ale RBFS rozwija węzły począwszy od najlepszych, nawet przy niedopuszczalnej heurystyce. Pomysł śledzenia najlepszej ścieżki alternatywnej pojawił się wcześniej w eleganckiej implementacji A* w języku Prolog (Bratko, 2009) oraz w algorytmie DTA* (Russell i Wefald, 1991) — w tej ostatniej pracy omawiane są także metapoziomowe przestrzenie stanów i metapoziomowe uczenie maszynowe.

Algorytm MA* pojawił się w pracy Chakrabarti i in. (1989); jego uproszczona wersja — SMA* — jest wynikiem wysiłków zmierzających do jego zaimplementowania (Russell, 1992). Kaindl i Khorsand (1994) zastosowali SMA* do stworzenia dwukierunkowego algorytmu wyszukiwania, który okazał się znacznie szybszy od poprzedników. Korf i Zhang (2000) opisują podejście „dziel i zwyciężaj", a Zhou i Hansen (2002) wprowadzają ograniczone pamięciowo grafowe wyszukiwanie A* praz strategię jego przełączania się na wyszukiwanie wszerz w celu bardziej efektywnego wykorzystywania pamięci (Zhou i Hansen, 2006).

Pomysł wyprowadzania dopuszczalnych heurystyk z rozluźnionych wersji oryginalnego problemu pojawił się w przełomowej pracy Helda i Karpa (1970), którzy zastosowali heurystykę minimalnego drzewa rozpinającego do rozwiązania problemu komiwojażera (patrz Ćwiczenie online 3.MSTR.). Automatyzacja procesu „rozluźniania" problemów została pomyślnie zaimplementowana przez Prieditisa (1993). Pojawia się coraz więcej literatury na temat zastosowania uczenia maszynowego do konstruowania heurystyk (Samadi i in., 2008; Arfaee i in., 2010; Thayer i in., 2011; Lelis i in., 2012).

Wykorzystanie baz wzorców do wyprowadzenia dopuszczalnych heurystyk jest zasługą Gassera (1995) oraz Culbersona i Schaeffera (1996, 1998); problem rozłączności baz wzorców opisywany jest przez Korfa i Felnera (2002). Wykorzystywanie wzorców symbolicznych jest pomysłem Edelkampa (2009). Felner i in. (2007) omawiają *kompresję* baz wzorców w celu oszczędniejszego wykorzystywania pamięci. Probabilistyczną interpretację heurystyki badali Pearl (1984) oraz Hansson i Mayer (1989).

Wśród podręczników traktujących o heurystykach w algorytmach wyszukiwania na szczególną uwagę zasługują *Heuristics* Pearla (1984) i *Heuristic Search* Edelkampa i Schrödla (2012). Publikacje na temat nowych algorytmów wyszukiwania pojawiają się pod egidą *International Symposium on Combinatorial Search* (SoCS) i *International Conference on Automated Planning and Scheduling* (ICAPS), a także w ramach wielu konferencji poświęconych sztucznej inteligencji, między innymi AAAI oraz IJCAI. Można je także znaleźć na łamach znanych czasopism, takich jak „Artificial Intelligence" i „Journal ACM".

SKOROWIDZ

A

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

Q

R

S

Ś

T

U

W

Z

Sztuczna inteligencja: to się staje na naszych oczach!

Sztuczna inteligencja budzi zachwyt i kontrowersje. W porównaniu z innymi gałęziami nauki jest stosunkowo młoda: liczy około siedemdziesięciu lat, mimo że czerpie ze znacznie starszych idei. Jednak błyskawiczny rozwój sztucznej inteligencji i przeobrażanie osiągnięć nauki w działające technologie sprawiają, że wyrobienie poglądu na całokształt tej dziedziny jest trudnym zadaniem. Warto więc spojrzeć na historię rozwoju sztucznej inteligencji z perspektywy jej współczesnych osiągnięć i dzięki temu lepiej zrozumieć, czym ta nauka jest w swojej istocie i dokąd podąża.

Oto pierwszy tom dzieła, które stanowi inspirujące spojrzenie na sztuczną inteligencję. Jego zrozumienie nie wymaga wybitnej znajomości informatyki i matematyki. Książka jest wspaniałą syntezą wczesnych i późniejszych koncepcji, a także technik, przeprowadzoną we frameworku idei, metod i technologii. Zawiera ogrom rzetelnej wiedzy przekazanej w niezbyt sformalizowany sposób. Opisy, formuły matematyczne i algorytmy, pokazane w postaci czytelnego pseudokodu, cechują się przejrzystością i precyzją. Zaprezentowano tu wszystkie ważne idee i koncepcje sztucznej inteligencji, zgodnie z najnowszymi trendami i osiągnięciami.

W tomie pierwszym między innymi:

- koncepcje sztucznej inteligencji
- różne podejścia do rozwiązywania problemów z wykorzystaniem sztucznej inteligencji
- reprezentacja wiedzy i modelowanie, a także wyszukiwanie i planowanie
- wnioskowanie w warunkach niepewności
- podejmowanie złożonych decyzji, również w środowisku wieloagentowym

Stuart Russell jest profesorem na Uniwersytecie Kalifornijskim w Berkeley, dyrektorem Center for Human-Compatible AI i profesorem inżynierii z ramienia fundacji Smitha-Zadeha. Laureat wielu prestiżowych nagród. Autor kilkuset publikacji dotyczących sztucznej inteligencji.

Peter Norvig jest dyrektorem do spraw badań w Google i członkiem kilku amerykańskich stowarzyszeń akademickich. Był szefem Wydziału Nauk Obliczeniowych NASA Ames Research Center. Napisał kilka cenionych książek dotyczących praktycznych aspektów sztucznej inteligencji.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej! ▶
ISBN 978-83-283-7608-3
9 788328 376083
Cena: 169,00 zł
Pearson