3.1.5. Iteratiivinen syventäminen (Iterative Deepening)

Luvun 3.1.4 algoritmin soveltamisessa on vaikeaa löytää tarpeeksi muttei liian suuri yläraja l. Jätetään siis tämäkin etsintäalgoritmin huoleksi:

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution sequence
   inputs: problem, a problem

   for depth ← 0 to ∞ do
      if DEPTH-LIMITED-SEARCH(problem, depth) succeeds then return its result
   end
   return failure
```

Algoritmi 3: Iteratiivinen syventäminen [RN95, kuva 3.15].

Tämä algoritmi yhdistää lukujen 3.1.1 ja 3.1.3 hyvät puolet:

- Se on *täydellinen* ja *optimaalinen* kuten luvun 3.1.1 leveyssuuntainen etsintäkin.
- Sen muistintarve on yhtä vaatimaton kuin käyttämällään luvun 3.1.3 syvyyssuuntaisella etsinnälläkin.
- Se tekee tosin useampia etsintäaskeleita kuin luvun 3.1.1 leveyssuuntainen etsintä, koska sama etsintäpuun solmu *laajennetaan monta kertaa*:
  1. Ensin käydään läpi koko korkeuden 0 puu löytämättä vastausta. Siihen kuuluu $T_0 = 1$ solmua (eli vain juuri) yhtälön (7) nojalla.
  2. Sitten läpikäydään korkeuden 1 puun
  $$T_1 = \frac{b^2 - 1}{b - 1} = b + 1$$
  solmua, joista 1 (pelkkä juuri) tutkittiin (muttei vielä laajennettu) jo kerran vaiheessa 1.
  3. Sitten korkeuden 2 puu, jossa juuri käsitellään kolmannen kerran, sen lapset toistamiseen, ja lapsenlapset ensimmäistä kertaa.

Kun näin jatketaan ratkaisusyvyyteen $depth = d$ saakka, on käsiteltyjen solmujen kokonaismäärä

$$\begin{aligned}
\sum_{0 \leq k \leq d} T_k = \sum_{0 \leq k \leq d} \frac{b^{k+1} - 1}{b - 1} & \\
&= \frac{1}{b-1} \left(\left(\sum_{0 \leq k \leq d} b^{k+1} \right) - \left(\sum_{0 \leq k \leq d} 1 \right) \right) \\
&= \frac{1}{b-1} \left(b \left(\sum_{0 \leq k \leq d} b^{k} \right) - (d+1) \right) \\
&= \frac{bT_d - (d+1)}{b-1}. \qquad (8)
\end{aligned}$$

- Toisaalta yhtälöstä (8) nähdään, että tämä algoritmi on vain noin

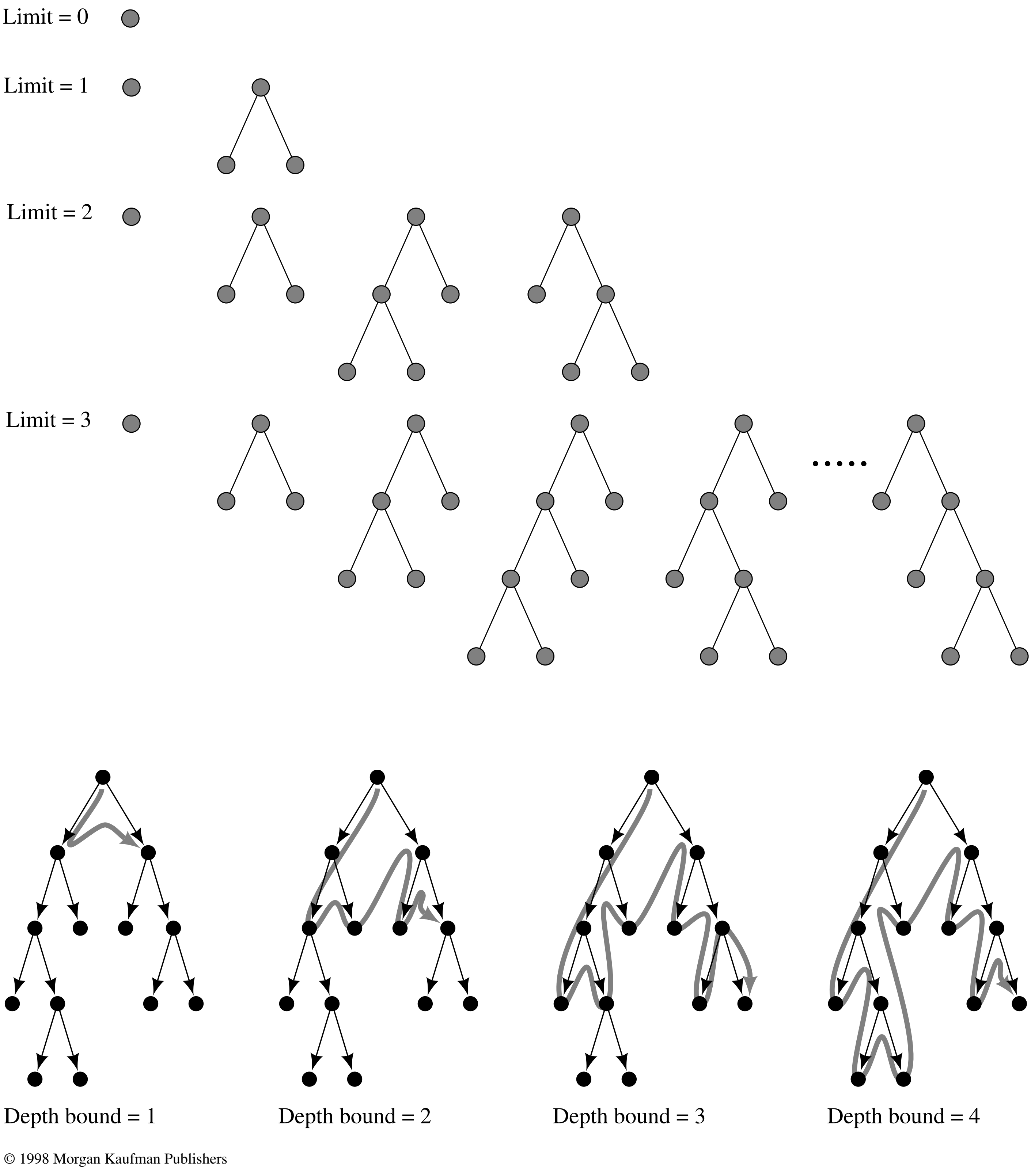
$$\frac{b}{b-1}$$

kertaa hitaampi kuin luvun 3.1.1 leveyssuuntainen etsintä:
  - Jos $b = 2$ niin se vie noin kaksinkertaisen ajan, mutta ero kutistuu kun b kasvaa.
  - Kun $b = 10$ ja $d = 5$ niin yhtälö (7) antaa $111\ 111$, ja yhtälö (8) antaa $123\ 456$, jolloin eroa on enää noin 11%.
  - Yleisestikin *suurin osa etsintäpuun solmuista on lehtiä*: jo binääripuissa on aina lehtiä 1 enemmän kuin sisäsolmuja, ja tämä ero vain kasvaa haarautumisasteen b mukana.
  - *Sisä*solmujen käsitteleminen moneen kertaan ei siis vaikuta merkittävästi kokonaistyömäärään.

Iteratiivinen syventäminen onkin paras systemaattinen etsintämenetelmä suurille avaruuksille, joiden halkaisijaa d ei tunneta etukäteen.

Jos samaa ideaa käytetään hintarajan ℓ määräämiseen, niin rajaa voi kasvattaa

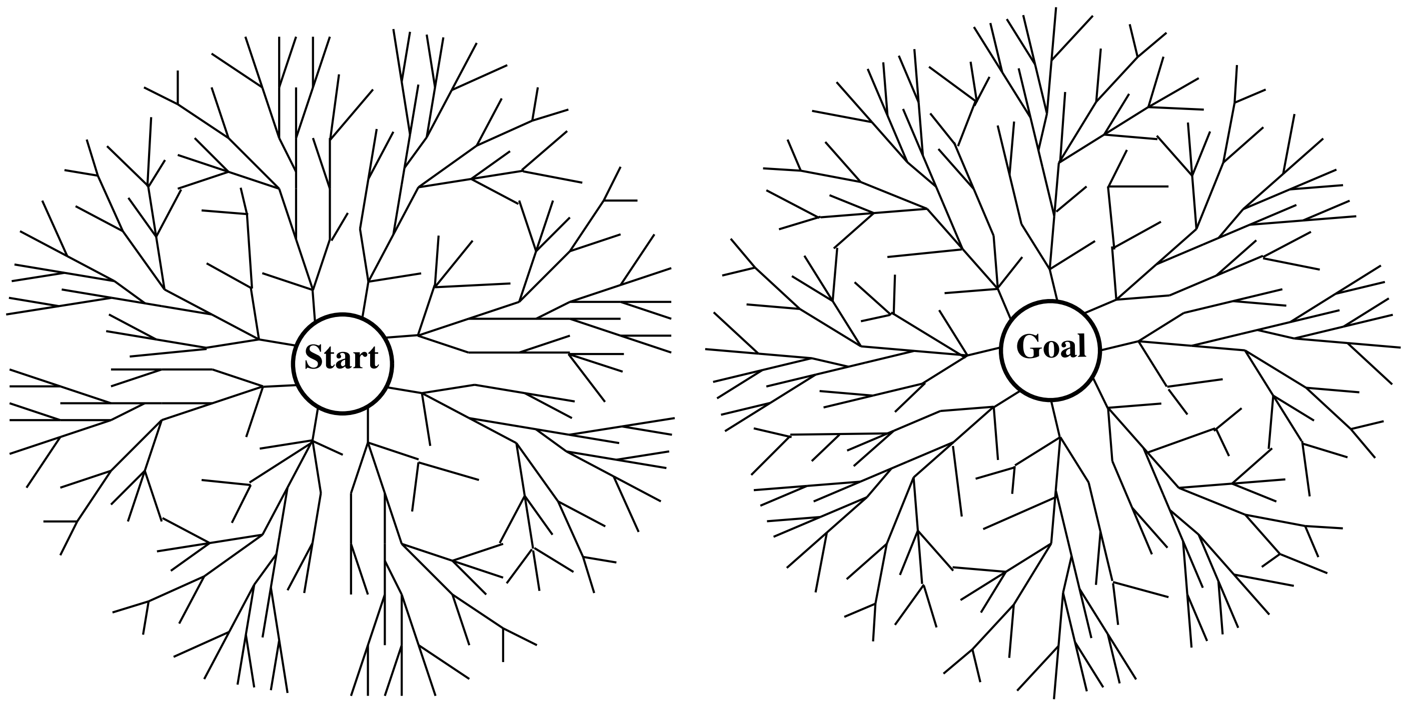
- joko systemaattisesti hinta-alarajan δ verran
- tai dynaamisesti pienimmäksi pituudeksi jolla jokin tavattu solmu jäi lapsettomaksi.



Kuva 25: Syventämisen eteneminen [RN95, kuva 3.16][Nil98, kuva 8.5].

3.1.6. Kaksisuuntainen etsintä (Bidirectional Search)

Entä jos käynnistetään yhtä aikaa yksi etsintä *eteenpäin alkutilasta* ja toinen etsintä *taaksepäin maalitilasta*, ja lopetetaan niiden kohdatessa?



Kuva 26: Kaksisuuntainen etsintä [RN95, kuva 3.17].

Periaatteessa voimme saavuttaa ratkaisun $O(2b^{d/2})$ askeleessa, koska kumpikin puolisko tulee toista puoliväliin vastaan.

Kun $b = 10$ ja $d = 6$, niin yhtälön (7) mukaan syvyyshaku laajentaa $1\ 111\ 111$ solmua, tämä algoritmi taas vain $2\ 222$.

Tekemällä 2 pienempää puuta karsimme lehtien määrää; luvun 3.1.5 havaiinon nojalla tämä kannattaa.

Ensin on kuitenkin ratkaistava seuraavat pulmat:

- Mitä tarkoittaa "etsiä taaksepäin"?
  Intuitiivisesti seurataan etsintäavaruuden nuolia takaperin.
- Onko siis jokaisella operaattorikaarellamme

$$P \xrightarrow{t} Q$$

vastakaari

$$P \xleftarrow{t^{-1}} Q$$

joka palauttaa tilanteen sellaiseksi, joka vallitsi ennen tekoa t?

Silloin nuolien seuraaminen takaperin voidaan tulkita etsinnäksi eteenpäin käyttäen vastatekoja t^{-1}.

Muuten voi olla vaikea määritellä miten nuolien seuraaminen takaperin oikein muuttaakaan etsinnän tilaa.

Esimerkiksi kuvan 15 kaltaisissa reititysongelmissa teko on peruutettavissa, ellei se kulje yksisuuntaista tietä.

- Entä jos maalitiloja on monta?

  Jos ne on annettu *eksplisiittisenä* äärellisenä listana, niin voidaan
  - etsiä taaksepäin ikään kuin se ongelma olisi monitilainen, tai
  - aloittaa yksi taaksepäin etsintä kustakin maalitilasta.

  Mutta jos onkin annettu *implisiittinen* maaliehdon kuvaus, niin pitäisi jotenkin järkeillä millaisista etsinnän tiloista saattaa seurata jokin maalitila. Tämä voi olla vaikeaa.

- Miten havaitaan, että etsinnät ovat kohdanneet?

  Aikaraja $O(b^{d/2})$ edellytti, että kohtaaminen voidaan tunnistaa vakioajassa. Tähän pyritään käytännössä usein *hajautustaululla (hash table)*.

- On valittava (ei välttämättä yksi ja sama) etsintäalgoritmi kummallekin suunnalle.

  Jos ainakin toinen näistä on luvun 3.1.1 leveyssuuntainen etsintä, niin kohtaamisen havaitseminen helpottuu.

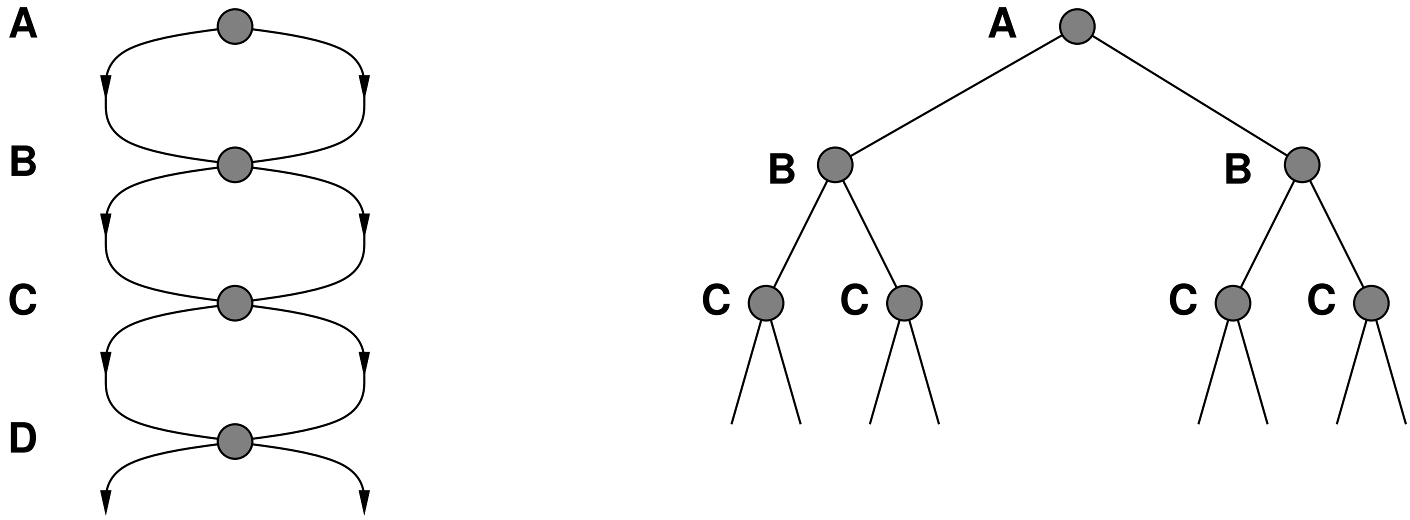
Verrataan lopuksi esiteltyjen algoritmien "strategisia mittoja":

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Time	b^d	b^d	b^m	b^l	b^d	$b^{d/2}$
Space	b^d	b^d	bm	bl	bd	$b^{d/2}$
Optimal?	Yes	Yes	No	No	Yes	Yes
Complete?	Yes	Yes	No	Yes, if $l \geq d$	Yes	Yes

Kuva 27: Systemaattisten etsintöjen vertailu [RN95, kuva 3.18].

3.1.7. Toistuvien tilojen eliminoimisesta

Etsinnän kuluessa saatamme löytää toistamiseen saman tilan, emmekä tietenkään halua tuhlata resursseja käsittelemällä sitä — ja etenkään siitä alkavaa koko alipuuta — uudelleen:



Kuva 28: Eksponentiaalinen eliminointi [RN95, kuva 3.19].

On ongelmia, joiden huolellinen määrittely ehkäisee tilojen toistumisen jo ennalta.

Toisaalta vaikkapa peruutettaville teoille näin ei ole.

Etsintäalgoritmia voidaan muuttaa siten, ettei se enää laajennakaan mitään sellaista etsinnän tilaa, joka on jo esiintynyt. . .

1. . . . sillä polulla, joka vie etsintäpuun nyt laajennettavasta lehtisolmusta sen juureen.

   Tämä polku on silloin pidettävä muistissa.

   Jokaista tilaa laajennettaessa sen polku on ensin käytävä läpi.

   Jos algoritmina on variantti luvun 3.1.3 syvyyssuuntaisesta etsinnästä, niin tämä polku löytyy pinosta.

2. . . . milloin tahansa etsinnän kuluessa.

   Muistintarve on O(avaruuden koko).

Hakurakenteena käytetään usein hajautusta.

Silti tämä vaihtoehto saattaa olla raskas.

Se on kuitenkin aggressiivisempi toiston torjuja kuin tapa 1.

Toisaalta eri poluilla olevien samojen tilojen rooli voi riippua myös polkuhinnasta: Jos kuvan 15 autoilutehtävässä minimoidaan vain kuljettujen teiden lukumäärää, niin reitit

(a) Arad – Sibiu – Rimnicu Vilcea – Craiova – Dobreta ja

(b) Arad – Timisoara – Lugoj – Mehadia – Dobreta

ovat "samat": molemmissa päädytään samaan kaupunkiin 4 askeleessa.

Jos taas minimoidaan kokonaismatkaa, niin reitti 2a on 486 km pitkä, mutta reitti 2b vain 374 km. Oletetaan, että reitti 2a löydetään ensin, ja sen jälkeen reitti 2b jätetään huomiotta "samana": saatamme päätyä väärään vastaukseen!

Silloin "samuus" on määriteltävä siten, että se ottaa huomioon myös kertyneen polkuhinnan.

Tällöin vähenee poistettavissa olevan toiston määrä.

Kokonaiskustannukset ja -hyödyt riippuvat tietenkin siitä, kuinka paljon toistoa etsintäongelmassamme esiintyy:

- Kuvan 20 mukaisessa 8-liukupelissä on $d \approx 20$ ja $b \approx 3$.
- Sen etsintäpuussa on siis noin 5.2×10^9 solmua yhtälöstä (7).
- Niistä vain $9! = 362\ 880$ on erilaisia.

  Polkuhinnoista (siirtomääristä) ei tarvinne pitää huolta.

  Näin pieni määrä mahtuu varmaankin jo keskusmuistiin.

  (Toisaalta virtuaalimuistiin joutuminen hidastaa.)
- Siis toistuvien tilojen poisto hävittää 99.99% etsintäpuusta!

  Eli kannattanee tässä tapauksessa.

3.2. Informoituja etsintämenetelmiä [RN95, luku 4] [Nil98, luku 9]

Luvun 3.1 algoritmit ovat usein tehottomia, koska ne eivät ota huomioon ratkaistavan etsintäongelman erityispiirteitä. Parempi tulos saadaan algoritmeilla, joita suunnittelija voi "ohjata" kohti tavoitetta jollakin *ongelmakohtaisella lisäinformaatiolla*.

Luvun 3.1.2 algoritmi käytti etsintäongelman määrittelyyn kuuluvaa polkuhintafunktiota g. Se valitsi aina laajennettavaksi sellaisen etsintäpuun solmun n, jossa "tähän mennessä toimijalle suunniteltu työmäärä"

$$g(n) = \text{juuresta solmuun } n \text{ vievän polun hinta}$$

on pienin. Lisäinformaatiota tuo puolestaan funktio

$$h^*(n) = \text{solmusta } n \text{ lähimpään ratkaisuun vievän polun hinta}$$

joka antaa "tästä eteenpäin jäljellä olevan työmäärän".

Tällaisten funktioiden ohjaama etsintämenetelmä on **"paras ensin" -etsintä (best-first search)**:

```
function BEST-FIRST-SEARCH(problem, EVAL-FN) returns a solution sequence
   inputs: problem, a problem
           Eval-Fn, an evaluation function

   Queueing-Fn ← a function that orders nodes by EVAL-FN
   return GENERAL-SEARCH(problem, Queueing-Fn)
```

Algoritmi 4: Paras ensin -etsintä [RN95, kuva 4.1].

Toteutus on kuten luvussa 3.1.2, paitsi että keko järjestetäänkin nyt ohjausfunktion EVAL-FN solmuille antamien arvojen mukaan.

Oikean funktion h^* *tarkkoja* arvoja ei yleensä ole tarjolla. Siksi käytetäänkin jotenkin etsintäongelmasta johdettua *heuristista arviofunktiota* h.

Sanalla "heuristiikka" (kr. *heurisko*, "keksiä"; vrt. "Heureka!") on useita eri merkityksiä:

- Tässä yhteydessä se tarkoittaa algoritmista tekniikkaa, joka saattaa *parantaa algoritmin suorituskykyä*.
- Se voi tarkoittaa myös sellaista yksinkertaistavaa oletusta, jonka avulla jokin vaikea *algoritminen ongelma helpottuu* — ratkeamattomasta ratkeavaksi, NP-täydellisestä polynomiseksi,. . .

  Esimerkiksi tyydytään tarkastelemaan vaikean yleisen tapauksen sijasta sellaisia yksinkertaisempia tapauksia jotka kattavat kaikki "käytännössä vastaan tulevat" tilanteet. . .
- Joskus se tarkoittaa jopa *algoritmisen ongelmanratkaisun vastakohtaa*, "luovaa inhimillistä" säännöistä vapaata ongelmanratkaisua!

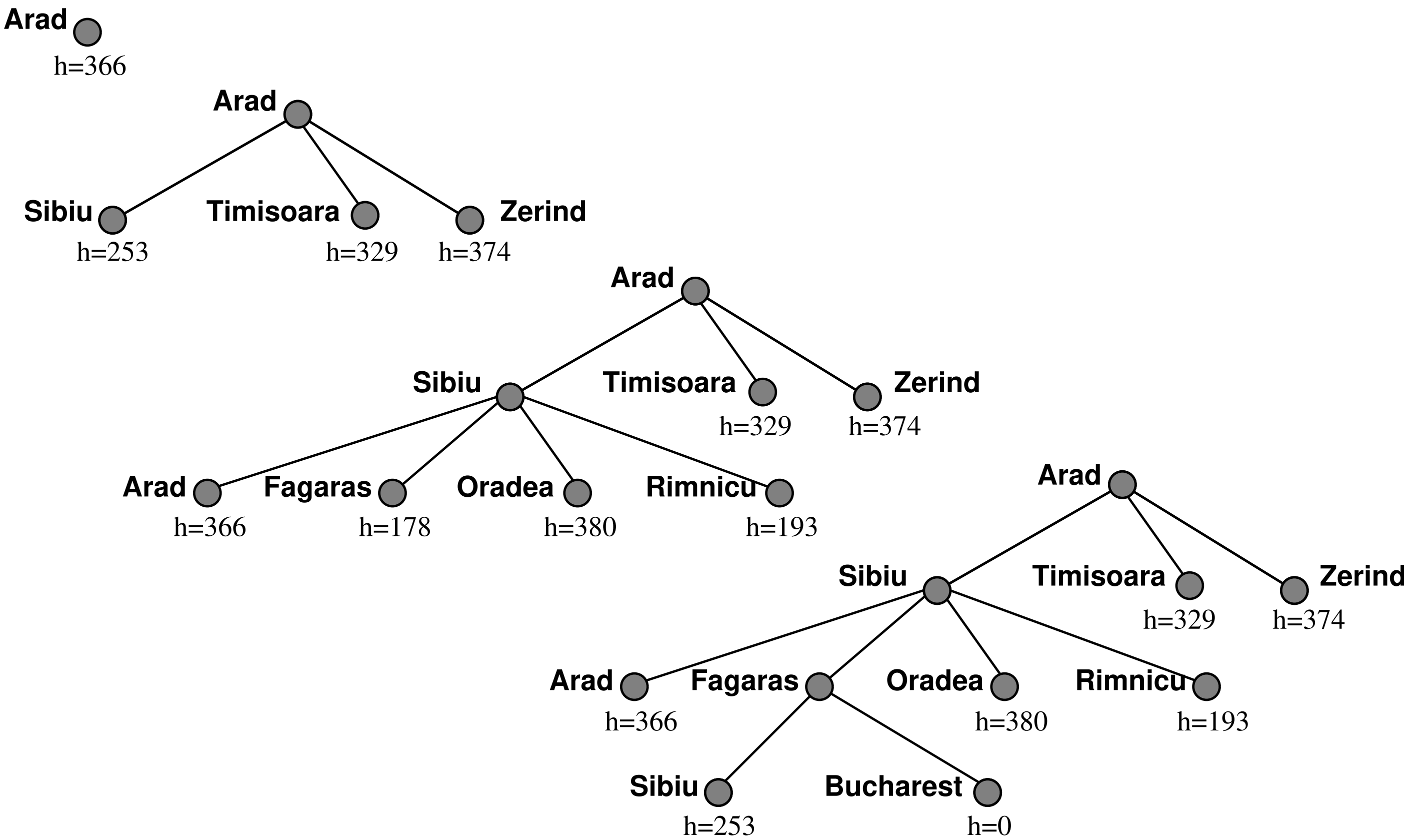
Yleisen systemaattisen etsintälgoritmin 2 tavoin yleisestä informoidusta etsintäalgoritmista 4 saadaan yksittäisiä algoritmeja kiinnittämällä laajennusjärjestyksen määräävä funktio $\textsc{Eval-Fn}$ eri tavoin.

Esimerkiksi luvun 3.1.2 algoritmi (joka ei vielä ollut informoitu eli ei käyttänyt funktiota h) saadaan valitsemalla $\textsc{Eval-Fn} = g$, eli ongelman polkuhinta juuresta nykyiseen lehtisolmuun.

Esimerkkinä informoidusta etsinnästä käytetään kuvan 15 tehtävää löytää reitti Aradista Bukarestiin. Lisäinformaatioksi h on kuvan oikeassa laidassa listattu kuinka kaukana Bukarest on kustakin kaupungista maantieteellisesti eli "linnuntietä". (Itse reitti kiemurtelee taas tarjolla olevia maanteitä.)

3.2.1. Ahne etsintä (Greedy Search)

Valitaan ensin Algoritmissa 4 $\textsc{Eval-Fn} = h$. Saatu algoritmi valitsee laajennettavaksi sen etsintäpuun lehden, joka kulloinkin *näyttää olevan lähimpänä tavoitetta*:



Kuva 29: Ahneesti Aradista Bukarestiin [RN95, kuva 4.3].

Tämä algoritmi on luvun 3.1.2 algoritmin "peilikuva":

- siellä pysyteltiin mahdollisimman lähellä lähtösolmua
- täällä rynnätään mahdollisimman lähelle maalia.

Ahne algoritmi käyttäytyy *usein muttei aina* ikään kuin luvun 3.1.3 syvyyssuuntainen etsintä, johon on lisätty "suuntavaistoa":

- Algoritmi pyrkii jatkamaan samaa polkua yhä eteenpäin ja eteenpäin, mutta nyt polkua ei jatketakaan "ensimmäisestä" lapsisolmusta vaan siitä — ei välttämättä (mutta usein) lapsi- — solmusta jonka arvioitu etäisyys tavoitteesta on pienin.

- Algoritmi muuttaa kulkuaan vasta kun tämä polku päätyy umpikujaan tai alkaa etääntyä tavoitteestaan. Mutta silloinkin peruutetaan yleensä samalla polulla taaksepäin ja kokeillaan jotakin melkein yhtä lupaavaa vielä tutkimatonta vaihtoehtoa.
- Kuvasta 29 nähdään, että ahneus *ei ole optimaalista*: löydetty reitti on 32 km pidempi kuin Rimnicu Vilcea – Pitesti -vaihtoehto. Tämä paras vaihtoehto jäi huomaamatta, koska Fagaras näytti *paikallisesti lupaavammalta*.
- Ahneus *ei ole myöskään täydellistä*: jos kuvan 15 ongelmana olisikin päästä Iasista Fagarasiin, niin ahne algoritmi (ilman luvun 3.1.7 toistojen poistoa) jää heilahtelemaan välille Iasi – Neamt.
- Algoritmien pahimman tapauksen aikavaatimukset ovat samat, joskin "hyvällä" h ahneesta algoritmista voi tulla hyvin nopea.
- Toisin kuin luvun 3.1.3 syvyyssuuntainen etsintä, ahne algoritmi joutuu pitämään kaikki etsintäpuun solmut muistissa.

  Etsintä saattaa näet palata myöhemmin kokeilemaan jotakin kesken jäänyttä solmua sitten kun kaikki sitä lupaavammat on käsitelty.

3.2.2. A*-etsintä

Luvun 3.2.1 ahne etsintä on nopea muttei täydellinen eikä optimaalinen. Luvun 3.1.2 algoritmi oli taas täydellinen ja optimaalinen muttei nopea. Yhdistetään ne (muista luku 3.2) valitsemalla Algoritmissa 4 $\textsc{Eval-Fn} = f$, missä

$$f(n) = g(n) + h(n) \tag{9}$$

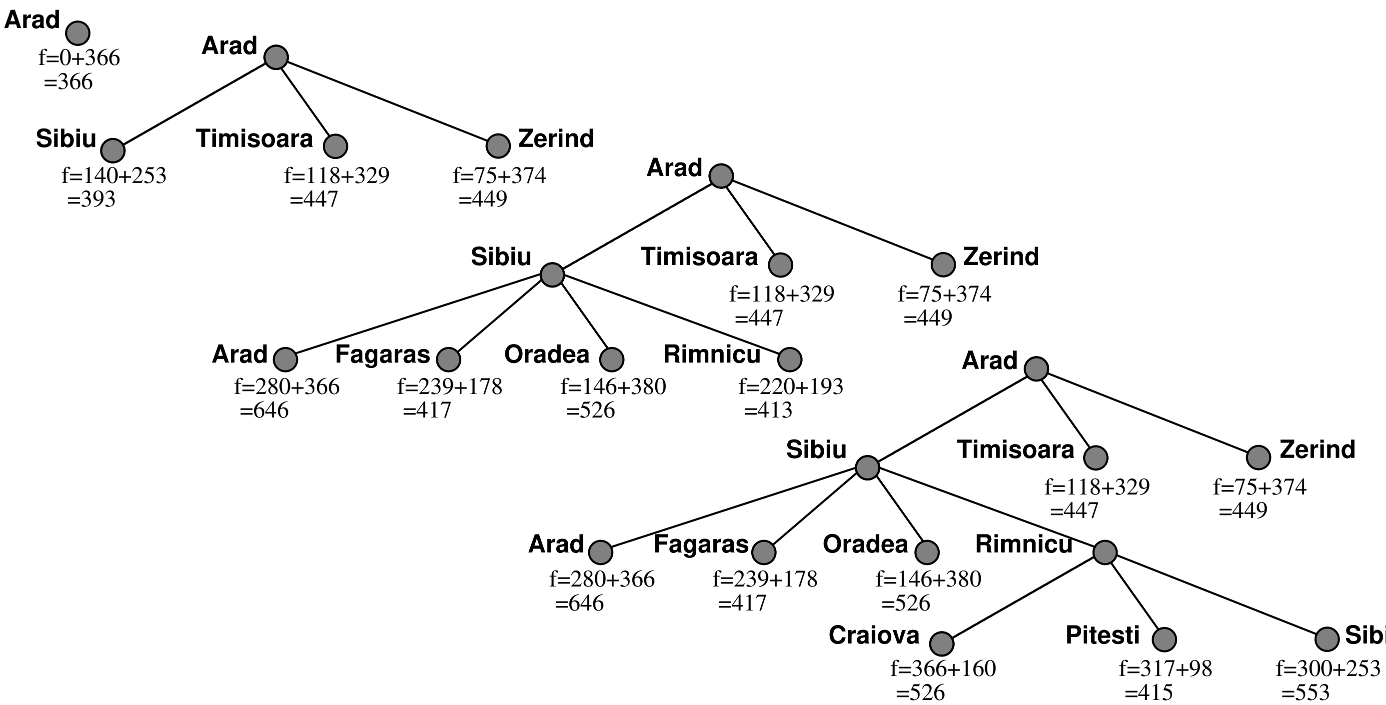
arvioi *halvimman sellaisen ratkaisun hintaa, joka kulkee solmun n kautta*.

Vaaditaan lisäksi että heuristiikka h on *luvallinen (admissible)*, eli että

$$h(n) \leq h^*(n)$$

jokaiselle solmuille n. Toisin sanoen,että h on *aina optimistinen*: vielä jäljellä oleva työmäärä näyttää aina pienemmältä kuin se oikeasti onkaan.

Silloin myöskään f ei koskaan yliarvioi pessimistisesti solmun n kautta kulkevien ratkaisujen hintoja. Kuvassa 15 luvallinen h on "etäisyys linnuntietä Bukarestiin":



Kuva 30: Parhaiten Aradista Bukarestiin [RN95, kuva 4.4].