monitilaiseen etsintäongelmaan ja kerätään jokainen vaihtoehtoinen Q_t yhteen kuvan 14 alapuolen tapaan.

Joskus tämä etsintäongelman osa muotoillaan **seuraajafunktio**na *(successor function)*

$$S(P) = \{\text{tila } Q_t \colon Q_t \text{ saadaan tilasta } P \text{ jollain teolla } t\} .$$

Etsintäavaruus *(search/state space)* on alkutilan ja operaattoreiden määräämä (mahdollisesti ääretön) suunnattu verkko, jonka

solmut ovat ne etsinnän tilat, joihin päästään alkutilasta jollakin jonolla tekoja, ja

kaaret ovat operaattorit $P \xrightarrow{t} Q_t$ kaikilla solmuilla P ja teoilla t

Esimerkkeinä olkoot kuvan 14 kaksi verkkoa.

Maaliehto *(goal test)* kertoo kustakin etsinnän tilasta, onko tavoite siinä saavutettu.

Etsintäongelman **ratkaisu** *(solution)* on mikä tahansa polku (tai sitä vastaava tekojono) sen alkutilasta johonkin maaliehdon täyttämään solmuun.

Polkuhinta *(path cost)* on funktio, joka laskee, paljonko annetun polun (= tekojonon) suorittaminen toimijalle "maksaisi" resursseina.

Tällä kurssilla koko polun

$$\mathcal{P} = Q_0 \xrightarrow{t_1} Q_1 \xrightarrow{t_2} Q_2 \xrightarrow{t_3} Q_3 \xrightarrow{t_4} \cdots \xrightarrow{t_n} Q_n$$

$$\text{hinta}(\mathcal{P}) = \sum_{1 \leq i \leq n} \text{hinta}(t_i)$$

on sen kunkin teon $\text{hinta}(t_i) > 0$ summa.

Kaikkiin ongelmiin ei liity hintoja: silloin valitaan $\text{hinta}(t) \equiv 1$ ja toivotaan lyhyttä ratkaisua.

Etsintäongelmien ratkaisumenetelmää M voidaan arvioida (ainakin) neljällä tavalla:

- Löytääkö M ratkaisun aina kun sellainen on olemassa?

Silloin M on *täydellinen (complete)*.

Vai voiko M epähuomiossa sivuuttaa ratkaisun

– ja pysähtyä vastaukseen "ei ole"
– tai olla pysähtymättä.

- Löytääkö M aina hinnaltaan halvimman ratkaisun *(online cost)*?

  Silloin M on *optimaalinen (optimal)* eli sen ehdottama tekojono on toimijalle "vähiten rasittava".

  Tai kuinka lähelle parasta ratkaisua M (yleensä) pääsee?

- Kuinka paljon *etsintä*resursseja (= muistia ja prosessoriaikaa) M kuluttaa *(offline cost)*?

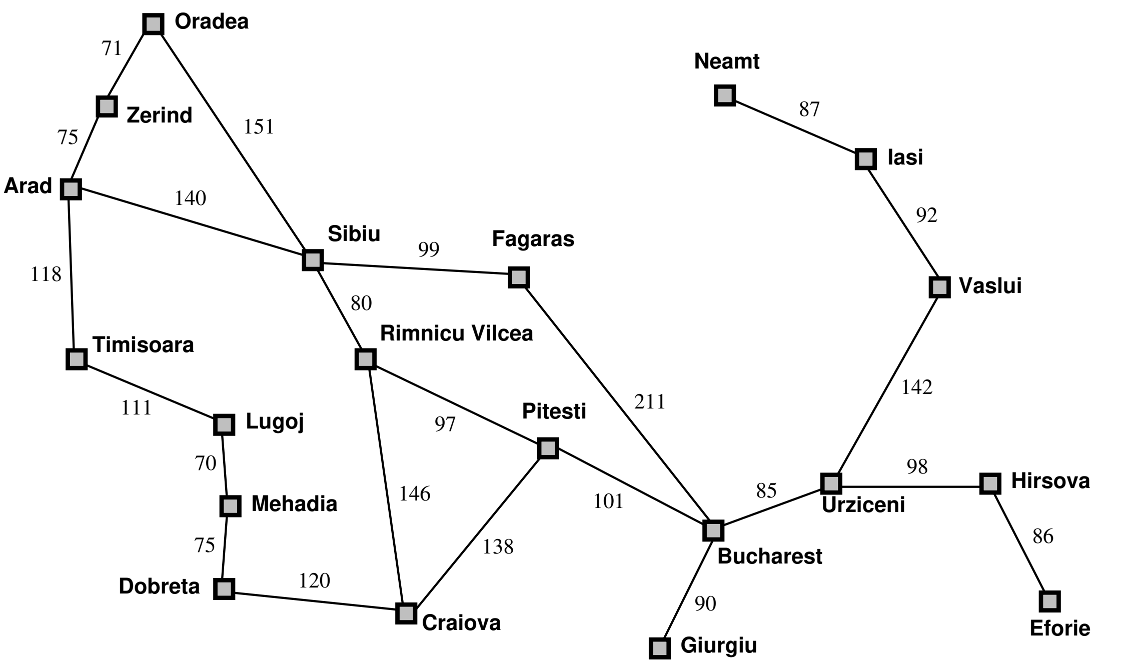
  Eli paljonko toimija käyttää voimiaan "miettimiseen"?

- Kuinka paljon M kuluttaa

$$\textit{kokonais}\text{resursseja} = \text{offline cost} + \text{online cost}$$

  (mahdollisesti painotettuna löydetun ratkaisun käyttökertojen lukumäärällä).

Tarkastellaan ongelmaa valita ajoreitti Aradista Bukarestiin.



Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

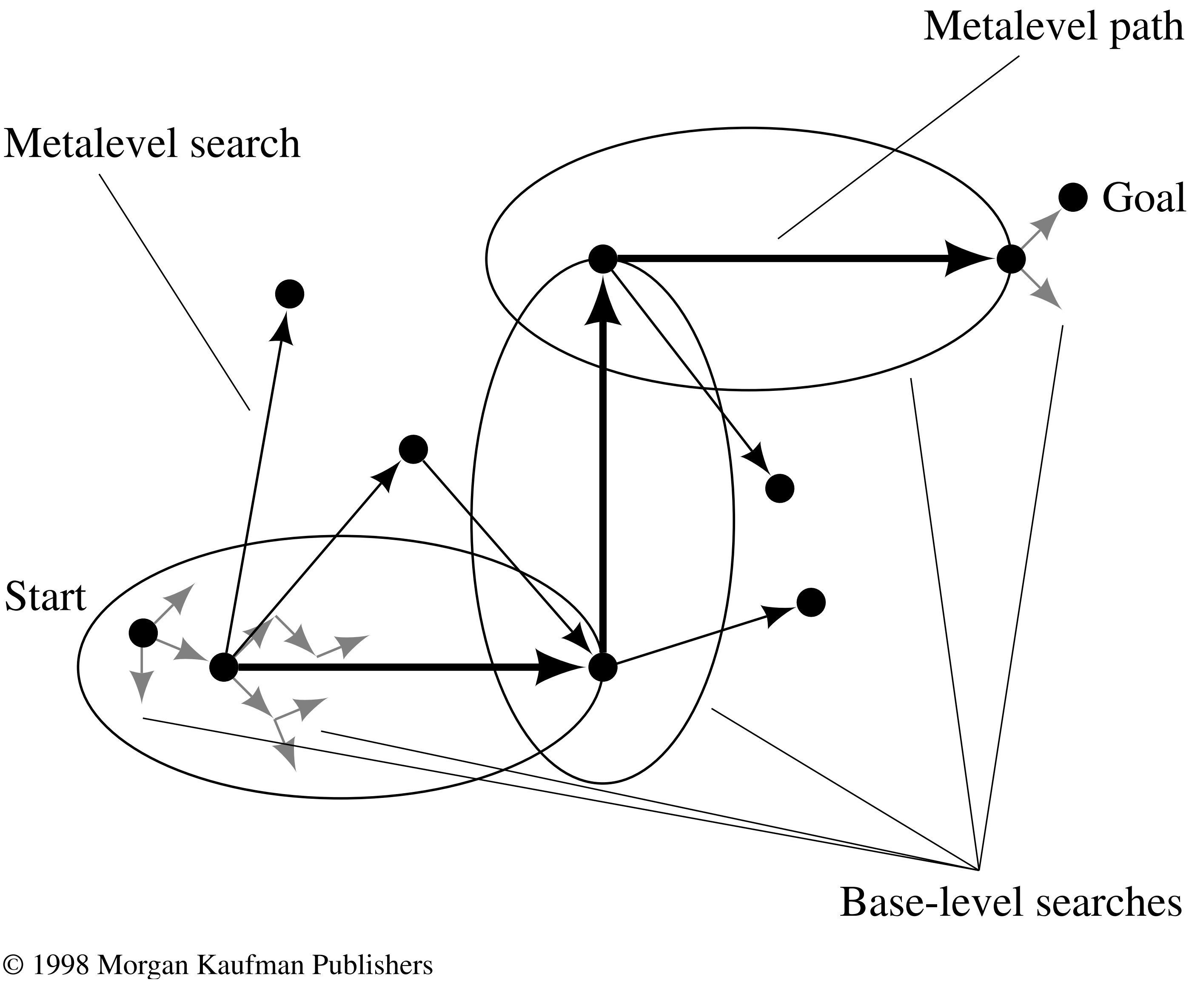
Kuva 15: Osa Romanian tieverkkoa. [RN95, kuva 4.2].

Alkutila on se, jossa olemme Aradissa.

Operaattorit valitaan viemään meidät kaupungista johonkin naapurikaupunkiin.

Tällä tasolla *pelkistämme (abstract)* ongelman kuvauksesta pois tarpeettomat ympäristön piirteet.

Pelkistämme pois myös tarpeelliset mutta liian yksityiskohtaiset piirteet: millaiset ohjaustoimenpiteet tarvitaan auton saamiseksi tästä parkkiruudusta kadulle, kadulta moottoritielle,. . . Ne voidaan selvittää *alemman pelkistystason* etsintäongelmana, kunhan ensin päätetään, mihin naapurikaupunkiin seuraavaksi suuntaamme. Tämä on *hierarkista* suunnittelua.



Kuva 16: Hierarkinen etsintä [Nil98, kuva 10.3].

Onnistunut abstraktio on välttämätöntä (muttei riittävää) onnistuneeseen etsintään!

Ja vaikeaa: pitäisikö esimerkiksi ottaa mukaan autokorjaamojen määrä kullakin reitillä — varmuuden vuoksi?

Maaliehto on, että olemme Bukarestissa.

Polkuhinta valitaan kuvan 15 informaation perusteella lyhimmäksi kokonaismatkaksi.

Ratkotaan tällaisia etsintäongelmia seuraavalla yleisellä algoritmi(skeema)lla:

```
function GENERAL-SEARCH(problem, QUEUING-FN) returns a solution, or failure

  nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
  loop do
      if nodes is empty then return failure
      node ← REMOVE-FRONT(nodes)
      if GOAL-TEST[problem] applied to STATE(node) succeeds then return node
      nodes ← QUEUING-FN(nodes, EXPAND(node, OPERATORS[problem]))
  end
```

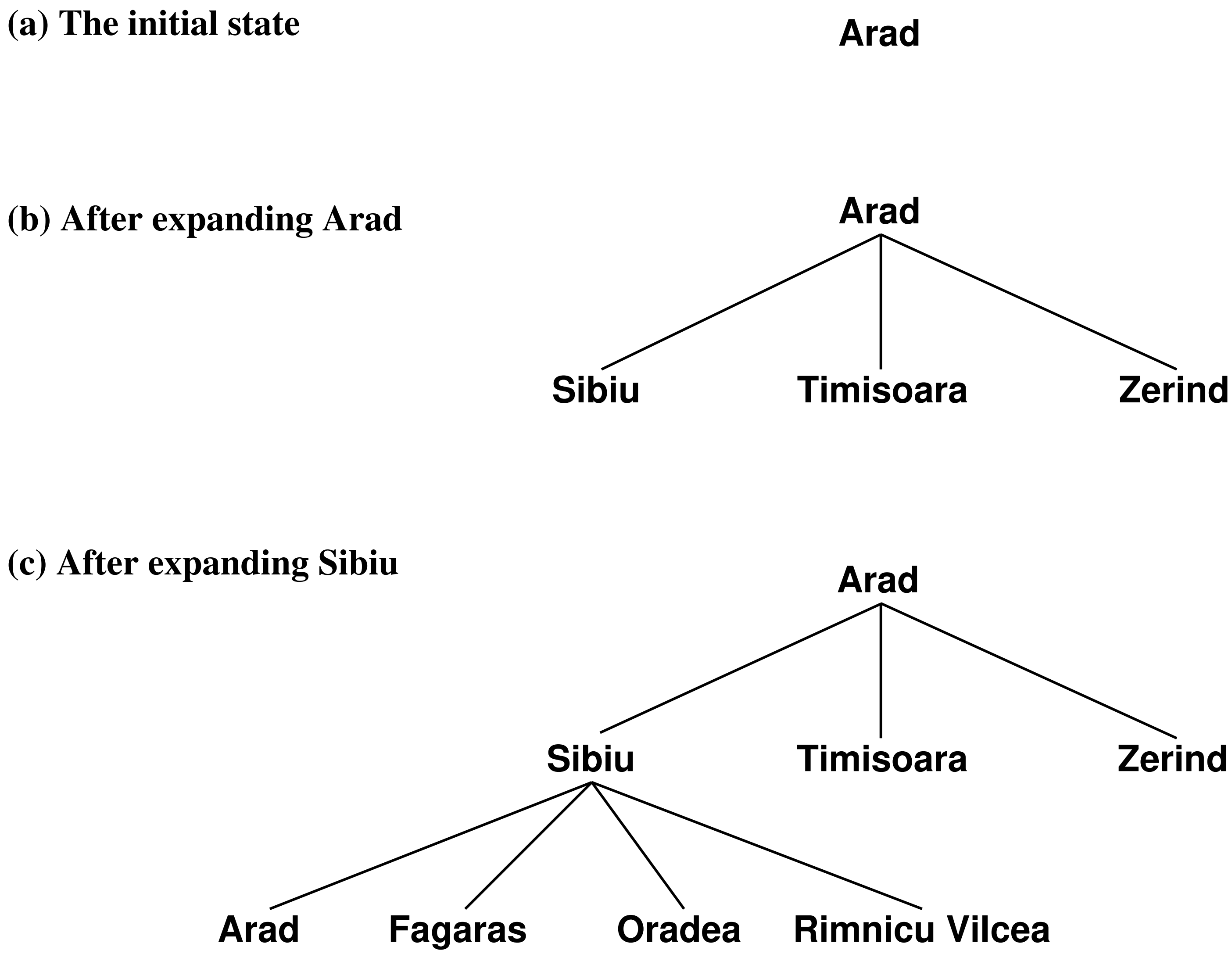
Algoritmi 2: Yleinen etsintäalgoritmi [RN95, kuva 3.10].

Parametrina saatu aliohjelma QUEUEING-FN tallettaa *työmuistiin* nimeltä *nodes* etsinnän eri vaihtoehtoja, joista REMOVE-FRONT poimii aina seuraavan tarkasteltavaksi.

Työmuistin ominaisuuksien valinnalla päätetään *etsintästrategia (search stratgy)* eli se järjestys, jossa eri vaihtoehtoja harkitaan.

Sovelletaan etsintäalgoritmia 2 ongelmaamme:

1. Aloitamme Aradista (kuva 17a), joka ei ole Bukarest, joten se on *laajennettava (expand)* naapureikseen (kuva 17b).
2. Strategiamme valitkoon naapurikaupungeista Sibiun kokeiltavaksemme (kuva 17c).
3. Jatkamme siitä kuten askeleessa 1, jne.



Kuva 17: Etsintäpuun kasvu. [RN95, kuva 3.8].

Strategian jäljittäminen tuottaa *etsintäpuun (search tree)*:

juuri on alkutila

solmut vastaavat etsintäavaruuden *polkuja*

lapset saadaan laajentamalla isäsolmun polkua yhdellä teolla

lehdistä valitaan yksi kullakin kierroksella laajennettavaksi strategian määräämässä järjestyksessä.

- Etsintäpuu kertoo, miten annettu etsintäalgoritmi käyttäytyy annetussa etsintäavaruudessa.

  Se on rekursiokutsupuun yleistys tapaukseen, jossa vaihtoehtojen käsittelyjärjestys seuraa työmuistin toiminnasta (ohjelmakoodin sijaan).

- Sen ei tarvitse olla algoritmia suorittavan tietokoneen muistissa:
  - jo käsiteltyä ei ole pakko muistaa
    (vaikkapa jos sen mahdollinen uudelleen käsittely ei haittaa)
  - vielä käsittelemätöntä saa unohtaa
    (vaikkapa jos se voidaan palauttaa takaisin käsiteltäväksi myöhemmin).
- Se voi olla *ääretön*, jos
  - avaruus on ääretön
    (jo ympäristö voi olla ääretön)
  - avaruudessa on kehä.

Jatkossa tarkastellaan erilaisia etsintäalgoritmeja antamalla vastaava QUEUEING-FN ja tarkastelemalla näin syntyviä etsintäpuita.

3.1. Systemaattisia etsintämenetelmiä [RN95, luku 3] [Nil98, luku 8]

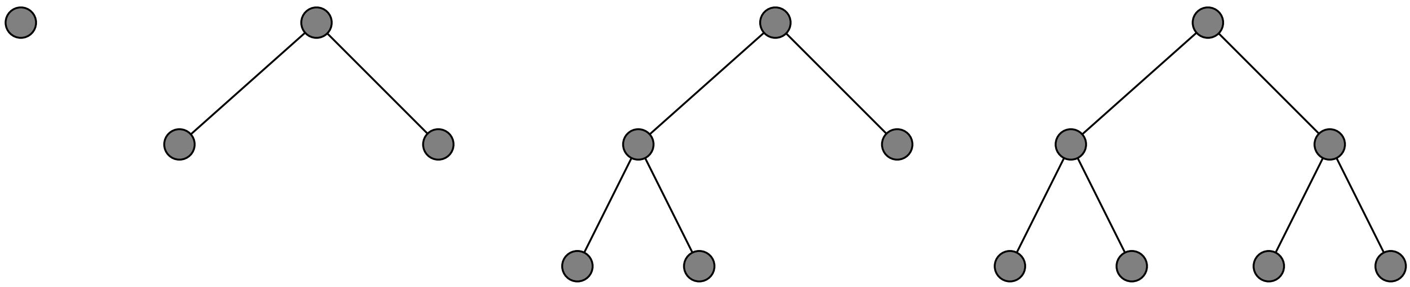
> "Ihaa, joka on ystäväni, on kadottanut häntänsä. Ja hän on ruvennut Murjottamaan. Voisitko ystävällisesti neuvoa minulle miten löytäisin sen?"
>
> "Hmm", sanoi Pöllö. "Tällaisissa tapauksissa Tavanomainen Proseduuri lienee seuraava."
>
> "Mitä tarkoittaa Tavanomainen Proosakuuri?" kysyi Puh. "Sillä minä olen Pieniälyinen Karhu jolle kummalliset sanat ovat Harmi."
>
> "Se tarkoittaa Mitä Tehdään."
>
> "Hyvä jos se tarkoittaa sitä", sanoi Puh nöyrästi.
>
> "Mitä Tehdään lienee seuraava: täytyy julistaa systemaattiset etsinnät. Sitten —"
>
> "Hetki vain", sanoi Puh käpälä pystyssä. "*Mitä* täytyy tehdä? Aivastelit juuri kun olit sanomaisillasi sen."
>
> *A.A. Milne: Nalle Puh*

Etsintämenetelmä on systemaattinen, jos se käy etsintäpuuta läpi jossakin systemaattisessa järjestyksessä hyödyntämättä mitään tietoa itse etsintäongelmasta. Sellaista etsintää kutsutaan myös *sokeaksi (uninformed/blind search*.

Vaikka systemaattinen etsintä onkin tehotonta, niin joskus ongelmastakaan ei tiedetä mitään.

3.1.1. Leveyssuuntainen etsintä (Breadth-First Search)

Jos Algoritmin 2 QUEUEING-FN on *jono (queue)*, niin saadaan (tunnetusti) etsintäpuun leveyssuuntainen läpikäynti:



Kuva 18: Leveyssuuntainen etsintä [RN95, kuva 3.11].

- Se on *täydellinen*.
- Se on myös *optimaalinen, jos* polkuhinta on solmun syvyyden ei-vähenevä funktio.

  Näin on kun kaikki operaattorit maksavat yhtä paljon.
- Ongelmana on *eksponentiaalinen (ajan ja) tilan tarve*:
  - Olkoon $b > 1$ avaruuden solmujen (suurin) *haarautumisaste (branching factor)* (= naapurien lukumäärä).
  - Olkoon lyhimmän ratkaisun (= polun) pituus eli avaruuden *halkaisija (diameter)* d tekoa.
  - Läpikäydyn puun koko (= laajennusten määrä) on silloin enintään

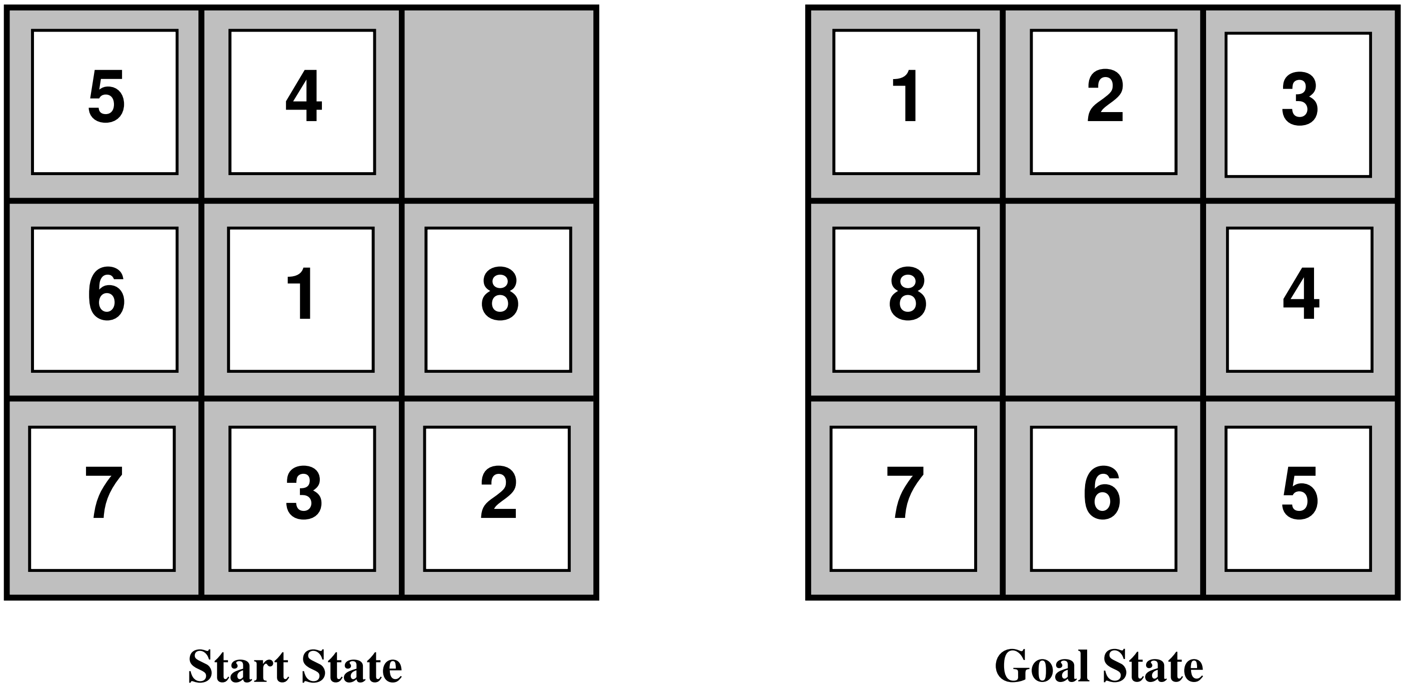
$$T_d = \sum_{0 \leq k \leq d} b^k = \frac{b^{d+1} - 1}{b - 1} \tag{7}$$

  solmua, joista enintään b^d on kerrallaan jonossa. (Geometrinen sarja [Myr74b, luku 11.3].)

Depth	Nodes	Time	Memory
0	1	1 millisecond	100 bytes
2	111	.1 seconds	11 kilobytes
4	11,111	11 seconds	1 megabyte
6	10^6	18 minutes	111 megabytes
8	10^8	31 hours	11 gigabytes
10	10^{10}	128 days	1 terabyte
12	10^{12}	35 years	111 terabytes
14	10^{14}	3500 years	11,111 terabytes

Kuva 19: Leveysetsinnän resurssit ($b = 10$) [RN95, kuva 3.12].

Eräs tekoälyn (opetuksen) suosittu leluongelma on lasten liukupeli:

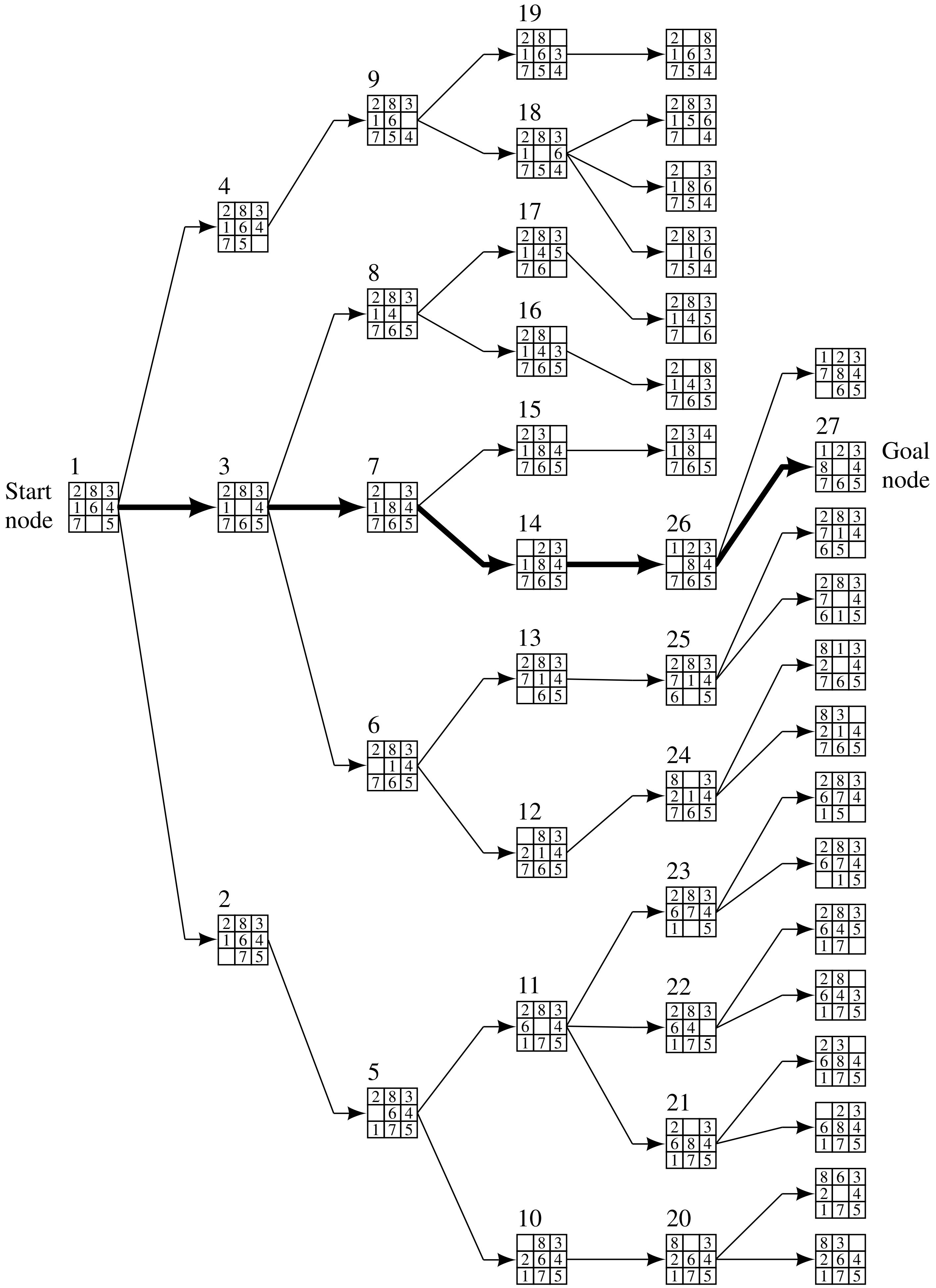


Kuva 20: 8-liukupeli [RN95, kuva 4.7].

Esimerkiksi kuvan 20 vasemmassa tilanteessa sen oikean yläkulman tyhjään ruutuun voi (yhtenä tekona) liu'uttaa

- joko vasemmanpuolisen naapurin $\boxed{4}$
- tai alapuolisen naapurin $\boxed{8}$

ja tyhjä paikka siirtyy sen tilalle.

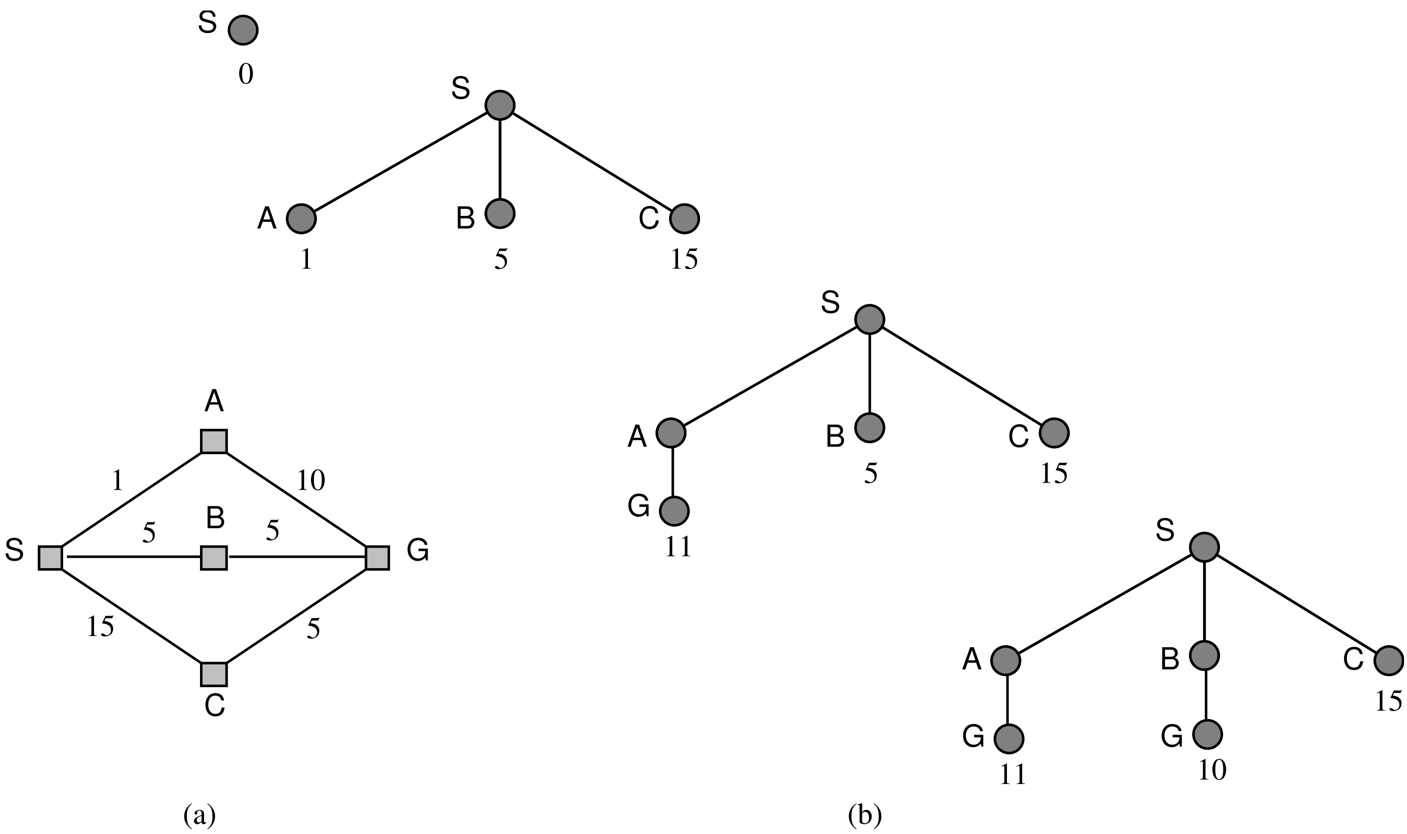


Kuva 21: Leveysetsintä 8-liukupelissä [Nil98, kuva 8.2].

3.1.2. Halvin ensin (Uniform Cost Search)

Luvun 3.1.1 algoritmi löytää sellaisen ratkaisun, jossa on *vähiten tekoja*, mutta jos eri teot ovat eri hintaisia ei se ole välttämättä *paras* valinta.

Algoritmin 2 QUEUEING-FN olkoon siis *keko (heap)* kuhunkin solmuun liittyvän polun (= tekojonon) kokonaiskustannuksen suhteen, joista valitaan aina pienin. Etsitään vaikkapa reitti solmusta S solmuun G:



Kuva 22: Halvin reitti ensin [RN95, kuva 3.13].

Tämä etsintäalgoritmi on *täydellinen ja optimaalinen, jos* polun kokonaishinta ei koskaan laske sitä myöten edettäessä.

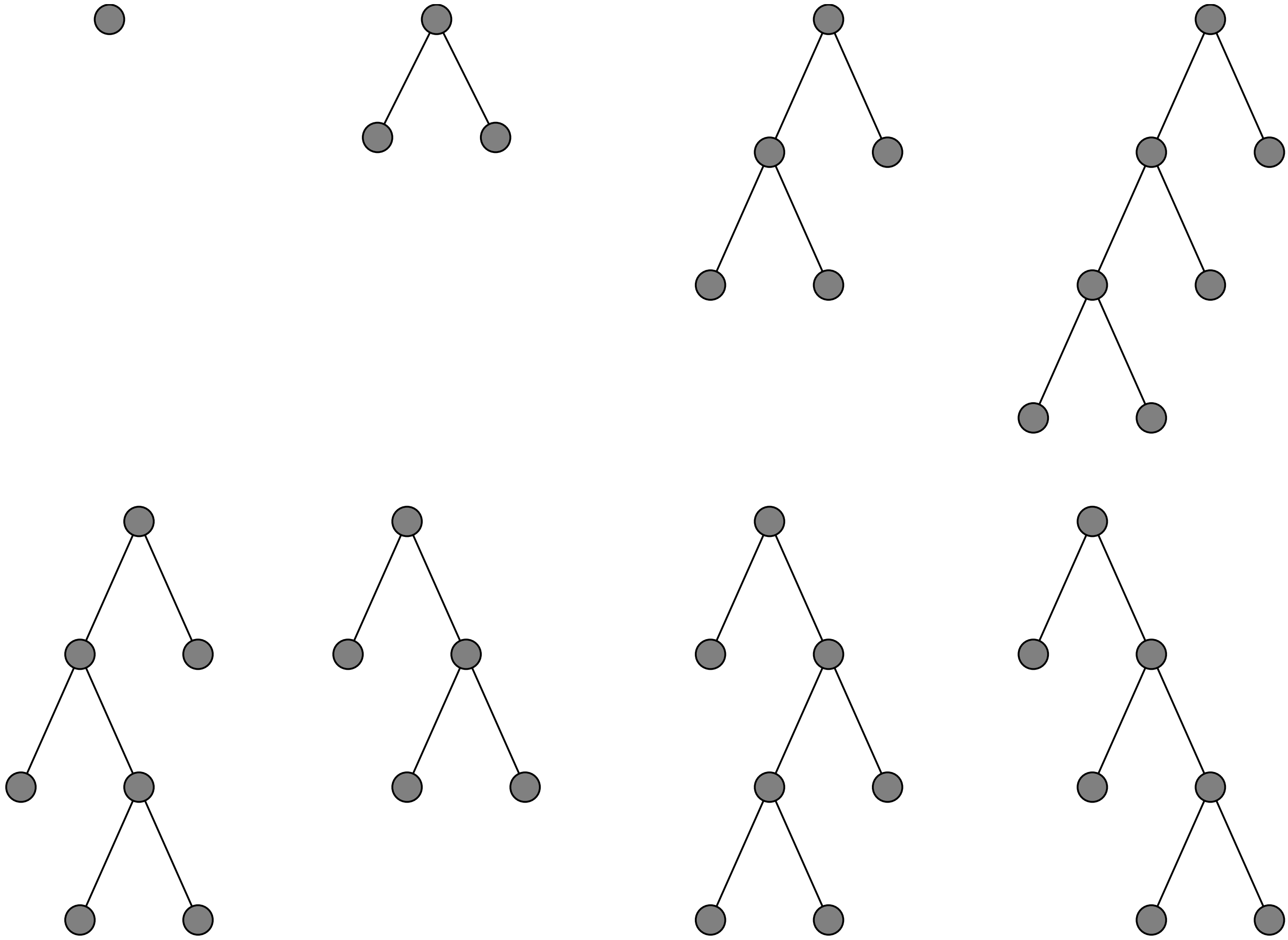
Luentojemme summautuvilla polkuhinnoilla tämä ehto on

$$\mathrm{hinta}(t) \geq 0$$

eli että mikään teko t ei tuota "voittoa" tekijälleen. Siis se on voimassa.

3.1.3. Syvyyssuuntainen etsintä (Depth-First Search)

Jos Algoritmin 2 QUEUEING-FN on *pino (stack)*, niin saadaan (tunnetusti) etsintäpuun syvyysyyntainen läpikäynti:



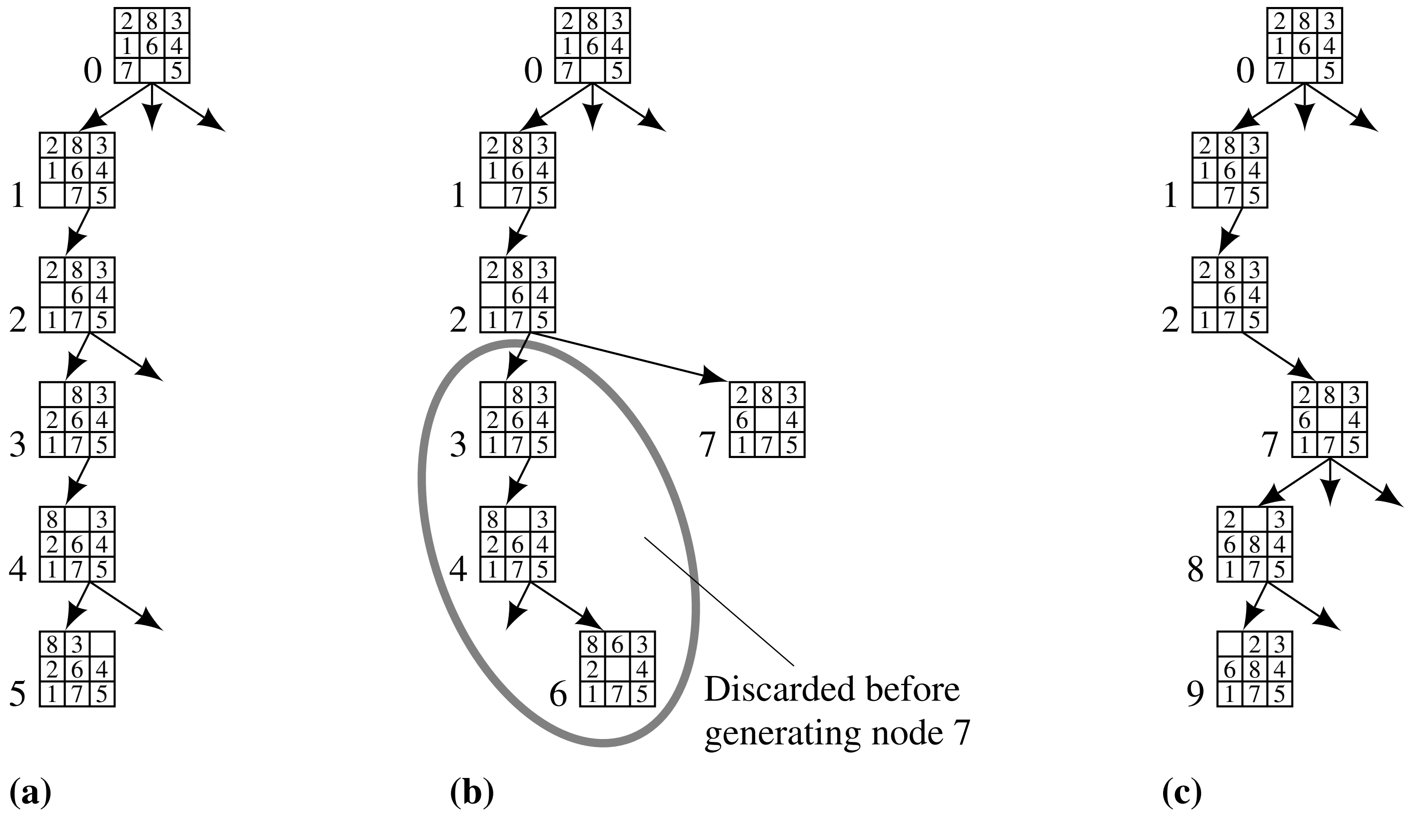
Kuva 23: Syvyyssuuntainen etsintä [RN95, kuva 3.14].

Tämä etsintäalgoritmi tarvitsee *paljon vähemmän muistia* kuin luvun 3.1.1 menetelmä: jos m on etsintäavaruuden pisimmän tekojonon pituus (= etsintäpuun syvyys), niin pinossa on kerrallaan vain $O(bm)$ solmua.

Tämä algoritmi vie pahimmillaan $O(b^m)$ etsintäaskelta, mutta jos avaruudessa on monta maaliehdon täyttävää solmua, se saattaa löytää yhden käymättä läpi kuin murto-osan koko avaruudesta.

Tämä algoritmi *ei ole optimaalinen*, eihän se edes pidä kirjaa eri polkujen kustannuksista.

Tämä algoritmi *ei edes pysähdy, jos* m on ääretön: se seuraa ensimmäistä tapaamaansa etsintäpuun ääretöntä haaraa yhä syvemmälle ja syvemmälle.



Kuva 24: Syvyyssuuntainen etsintä 8-puzzlessa [RN95, kuva 8.3].

Jos etsintäongelman ratkaisijan ohjelmoi *rekursiivisena* niin päätyy johonkin syvyyshaun varianttiin: rekursiokutsupino toimii silloin (näkymättömänä) hakupinona.

Ja kääntäen: jos valitsee menetelmäkseen syvyyshaun, niin sen voi usein *toteuttaa helposti* rekursiolla.

3.1.4. Syvyysrajoitettu etsintä (Depth-Limited Search)

Yksi tapa saada luvun 3.1.3 algoritmi varmasti pysähtymään on lisätä siihen etukäteen yläraja l tarkasteltavien polkujen *pituudelle* (eli seurattavien haarojen syvyydelle): jos nykyinen solmu ylittää rajan, niin sitä ei enää laajennetakaan lapsikseen.

- Tämä syvyysrajoitettu etsintä ei edelleenkään ole optimaalinen.
- Jos $l \geq d$, niin se on täydellinen, muuten taas ei.
- Se vie $O\left(b^l\right)$ etsintäaskelta.
- Se pitää pinossaan korkeintaan $O(bl)$ solmua kerrallaan.

Samaa ideaa voi soveltaa myös antamalla yläraja ℓ polkujen *hinnalle*:

- Pysähtyminen on taattu jos ei ole ääretöntä polkua jonka hinta pysyisi silti äärellisenä.
  Luentojemme summatuvilla polkuhinnoilla tämä tarkoittaa
  $$\mathrm{hinta}(t) \geq \delta$$
  missä $\delta > 0$ on kaikkien tekojen hintojen alaraja.