Proiectarea Algoritmilor

Curs 11- Preflux (continuare).
Algoritmi euristici de explorare

Bibliografie

[1] Cormen – Introducere în algoritmi – cap Algoritmi de preflux (27.4)

[2] C. Giumale – Introducere in Analiza Algoritmilor - cap. 7

[3] http://www.gamasutra.com/features/19990212/pathdemo.zip

[4] http://www.policyalmanac.org/games/aStarTutorial.htm

[5] http://www.ai.mit.edu/courses/6.034b/searchcomplex.pdf

Reminder - Arc rezidual. Capacitate reziduală.

- Definiție: Un arc (u,v) pentru care f(u,v) < c(u,v) se numește arc rezidual.
- ➔Fluxul pe acest arc se poate mări.
- Definiție: Cantitatea cu care se poate mări fluxul pe arcul (u,v) se numește capacitatea reziduală a arcului (u,v) (c_f(u,v)):

$$c_f(u,v) = c(u,v) - f(u,v)$$

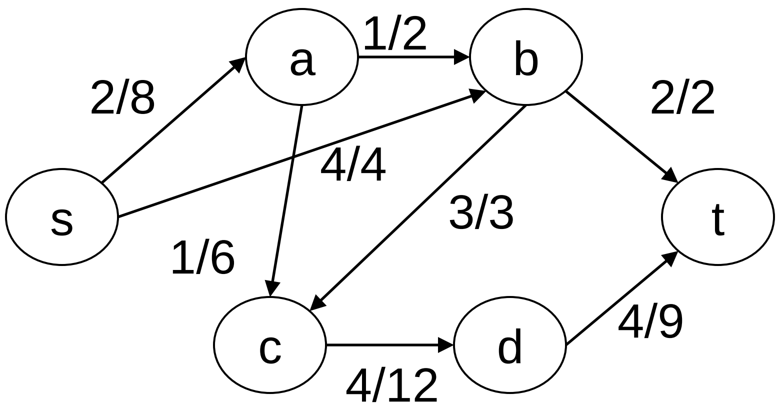
Reminder - Rețea reziduală. Cale reziduală.

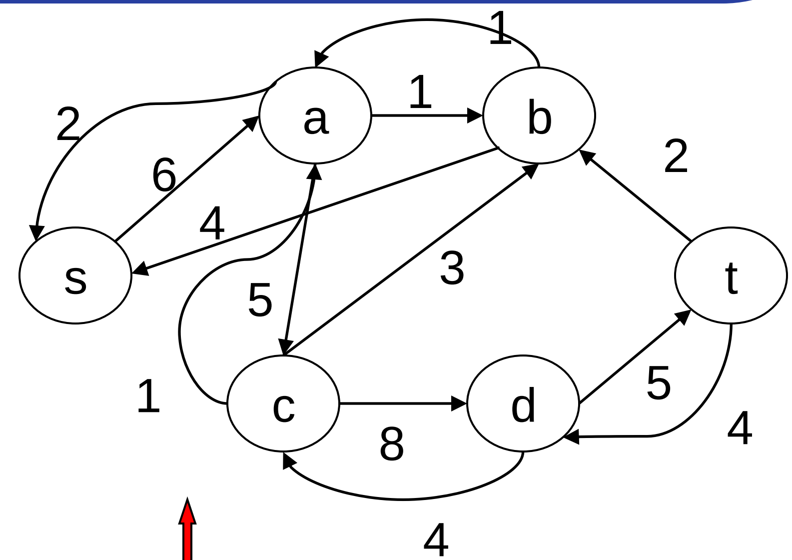
- $G = (V,E)$ rețea de flux cu funcția de capacitate c.
- Definiție: Rețeaua reziduală ($G_f = (V,E_f)$) este o rețea de flux formată din arcele ce admit creșterea fluxului:

$$E_f = \{(u,v) \in V \times V \mid c_f(u,v) > 0\}.$$

- Observație: $E_f \not\subseteq E$!!!
- Definiție: Cale reziduală e un drum s..t $\subseteq G_f$, unde $c_f(u,v)$ este capacitatea reziduală a arcului (u,v).
- Definiție: Capacitatea reziduală a căii = capacitatea reziduală minimă de pe calea s..t descoperită.

Reminder – Graf rezidual, rețea reziduală, capacitate reziduală

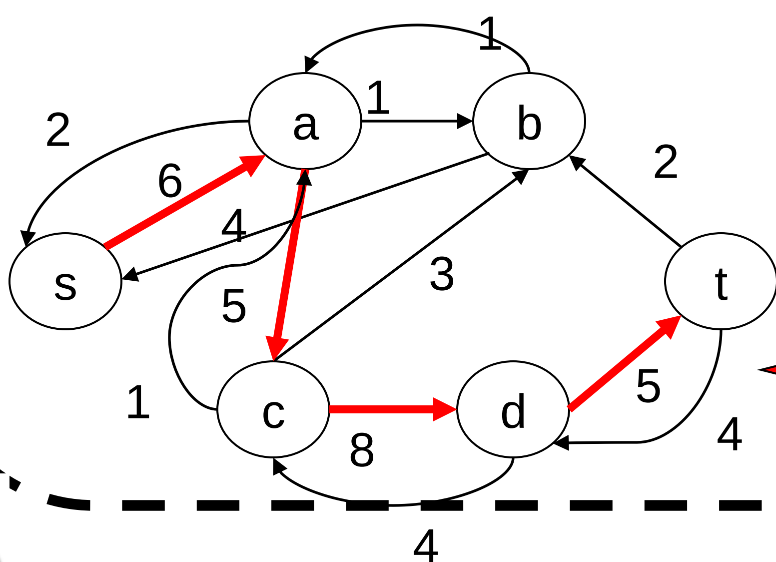




$c_f(u,v) = c(u,v) - f(u,v)$

Rețeaua reziduală $G_f = (V,E_f)$ unde

$E_f = \{(u,v) \in V \times V \mid c_f(u,v) > 0\}$



Calea reziduală: s → a → c → d → t

Capacitatea reziduală a căii:

$c_f(p) = \min\{6, 5, 8, 5\} = 5$

Pompare preflux (1)

- Idee: Simularea curgerii lichidelor într-un sistem de conducte ce leagă noduri aflate la diverse înălțimi;
- Sursa – înălțime maximă;
- Inițial toate nodurile exceptând sursa sunt la înălțime 0;
- Destinația rămâne în permanență la înălțimea 0!

Pompare preflux (2)

- Există un preflux inițial în rețea obținut prin încărcarea la capacitate maximă a tuturor conductelor ce pleacă din s;
- Excesul de flux dintr-un nod poate fi stocat într-un rezervor al nodului (Notat e(u));
- Când un nod u are flux disponibil în rezervor și o conductă spre un alt nod v nu este încărcată complet → înălțimea lui u este crescută pentru a permite curgerea din u în v.

Pompare preflux – Definiții (1)

- G = (V,E) rețea de flux;
- Definiție: Preflux = f: V x V → $\Re$ astfel încât să fie satisfăcute restricțiile:
  - $f(u,v) \leq c(u,v)$, $\forall(u,v) \in E$ – respectarea capacității arcelor;
  - $f(u,v) = -f(v,u)$, $\forall u,v \in V$ – simetria fluxului;
  - $\Sigma_{v \in V} f(v,u) \geq 0$, $\forall u \in V \setminus \{s\}$ – ~~conservarea fluxului.~~
- Definiție: Supraîncărcare a unui nod:
  - $e(u) = f(V,u) \geq 0$, $\forall u \in V \setminus \{s\}$.

Pompare preflux – Definiții (2)

- Definiție: O funcție h: V → N este o funcție de înălțime dacă îndeplinește restricțiile:
  - $h(s) = |V|$ – fixă;
  - $h(t) = 0$ – fixă;
  - $h(u) \leq h(v) + 1$ pentru orice arc rezidual $(u,v) \in G_f$ – variabilă.

- Lema 5.19: G – rețea de flux, h: V → N este o funcție de înălțime. Dacă $\forall u, v \in V$, $h(u) > h(v) + 1$ atunci arcul (u,v) nu este arc rezidual.

Pompare preflux – Metode folosite

- Pompare(u,v) // pompează fluxul în exces (e(u) > 0)
  // are loc doar dacă diferența de înălțime dintre u și v este 1
  // (h(u) = h(v) + 1), altfel nu e arc rezidual și nu ne interesează
  - d = min(e(u), c_f(u,v)); // cantitatea de flux pompată
  - f(u,v) = f(u,v) + d; // actualizare flux pe arcul (u,v)
  - f(v,u) = -f(u,v); // respectarea simetriei
  - e(u) = e(u) – d; // actualizare supraîncărcare la sursă
  - e(v) = e(v) + d; // actualizare supraîncărcare la destinație

- Înălțare(u) // mărește h(u) dacă u are flux în exces
  // (e(u) > 0) și u $\notin$ {s, t} $\forall$(u,v) $\in G_f$ avem h(u) ≤ h(v)
  - h(u) = 1 + min{h(v) | (u,v) $\in G_f$}

Pompare preflux – Inițializare

- Init_preflux(G, s, t)
  - **Pentru fiecare** ($u \in V$)
    - e(u) = 0 // inițializare exces flux în nodul u
    - h(u) = 0 // inițializare înălțime nod u
    - **Pentru fiecare** ($v \in V$) // inițializare fluxuri
      - f(u,v) = 0
      - f(v,u) = 0
  - h(s) = |V| // inițializare înălțime sursă
  - **Pentru fiecare** ($u \in succs(s) \setminus \{s\}$)

    // actualizare flux + exces
    - f(s,u) = c(s,u);
    - f(u,s) = -c(s,u);
    - e(u) = c(s,u)

Pompare preflux – Algoritm

- Pompare_preflux(G, s, t)
  - Init_preflux(G, s, t) // inițializarea prefluxului
  - **Cât timp** (1) // cât timp pot face pompări sau înălțări
    - **Dacă** ($\exists$u $\in$ V \ {s, t}, v $\in$ V | e(u) > 0 **și** c_f(u,v) > 0 **și** h(u) = h(v) + 1) // încerc să pompez
      - Pompare(u,v); ***continuă;***
    - **Dacă** ($\exists$u $\in$ V \ {s, t}, v $\in$ V | e(u) > 0 **și** $\forall$(u,v) $\in$ E_f, h(u) $\leq$ h(v))
      - Înălțare(u); ***continuă;*** // încerc să înalț
    - **Întrerupe;** // nu mai pot face nimic → am ajuns la flux max
  - **Întoarce** e(t) // e(t) = |f| = fluxul total în rețea

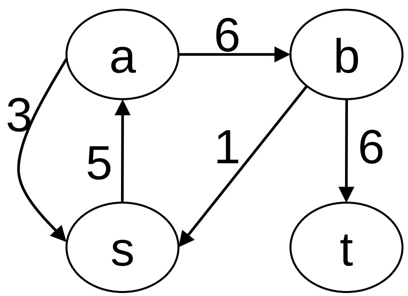
- Complexitate?

Pompare preflux – Complexitate

- Init_preflux: O(V * E)
- Pompare(u,v): O(1)
- Înălțare(u): O(V) – implică găsirea minimului dintre nodurile succesoare
- Cât timp: [vezi Cormen]
  - Câte înălțări?
    - Care e înălțimea maximă?
      - Lemă: Pt ∀ vârf excedentar u, ∃ un drum de la u la s în G_f → 2 |V| - 1
    - Care este numărul maxim total de înălțări? (2 |V| - 1) (|V| - 2)
  - Câte pompări?
    - Pompări saturate (prin care arcul respectiv devine saturat – c_f = 0):
    - Se calculează câte valori poate lua h(u)+h(v) și apoi se /2 (diferența min. de h dintre pompări) → 2 |V| |E|
    - Pompări nesaturate (restul):
      - Se calculează valoarea maximă a sumei înălțimilor nodurilor excedentare ($4 |V|^2 (|V| + |E|)$):
        - Înălțarea fiecărui nod poate adăuga 2 |V|
        - Fiecare pompare saturată adaugă 2 |V| (nodul către care se pompează poate deveni excedentar)
      - Fiecare pompare nesaturată scade minim 1 (h(u)=h(v)+1 și v devine excedentar) –> max $4 |V|^2 (|V| + |E|)$
- Complexitate totală: $O(V^2 * E)$ [vezi Cormen]

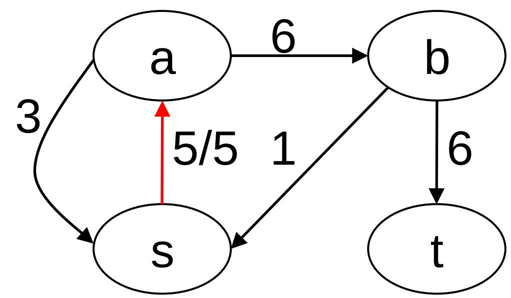
Exemplu Pompare preflux (1)

G

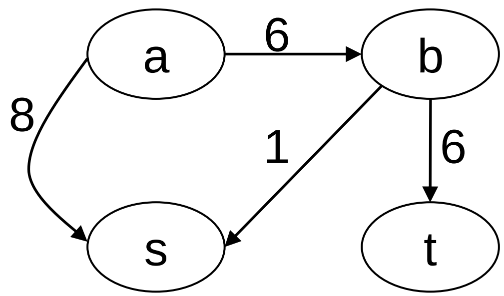


Init_preflux →

$h(s) = 4$
$h(a) = h(b) = h(t) = 0$
$e(a) = 5$
$e(s) = e(b) = e(t) = 0$



$\xrightarrow{G_f}$



G_f

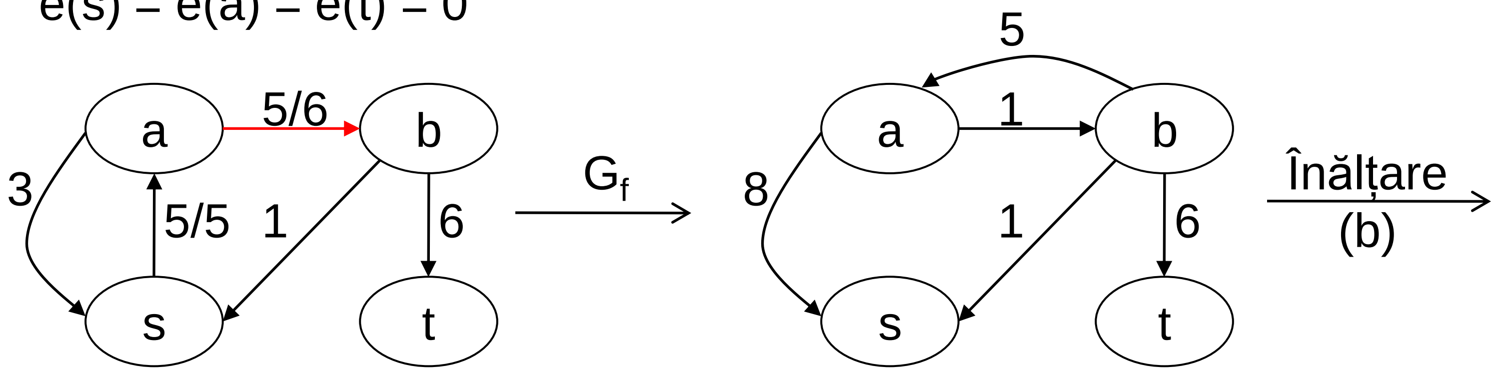
Înălțare (a) →

$h(s) = 4$
$h(a) = 1$
$h(b) = h(t) = 0$
$e(a) = 5$
$e(s) = e(b) = e(t) = 0$

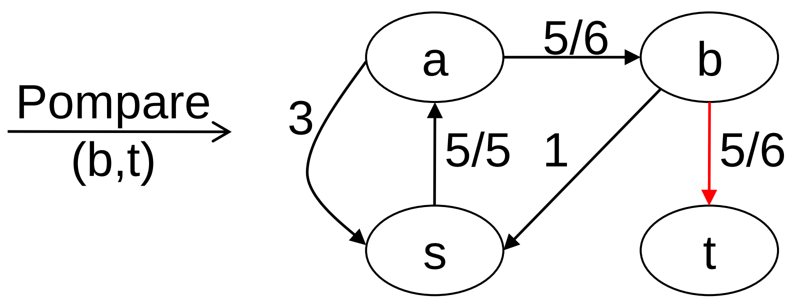
Pompare (a,b) →

Exemplu Pompare preflux (2)

h(s) = 4
h(a) = 1
h(b) = h(t) = 0
e(b) = 5
e(s) = e(a) = e(t) = 0



h(s) = 4
h(a) = h(b) = 1
h(t) = 0
e(b) = 5
e(s) = e(a) = e(t) = 0



h(s) = 4
h(a) = h(b) = 1
h(t) = 0
e(t) = 5
e(s) = e(a) =
= e(b) = 0

Proiectarea Algoritmilor

Algoritmi euristici de explorare

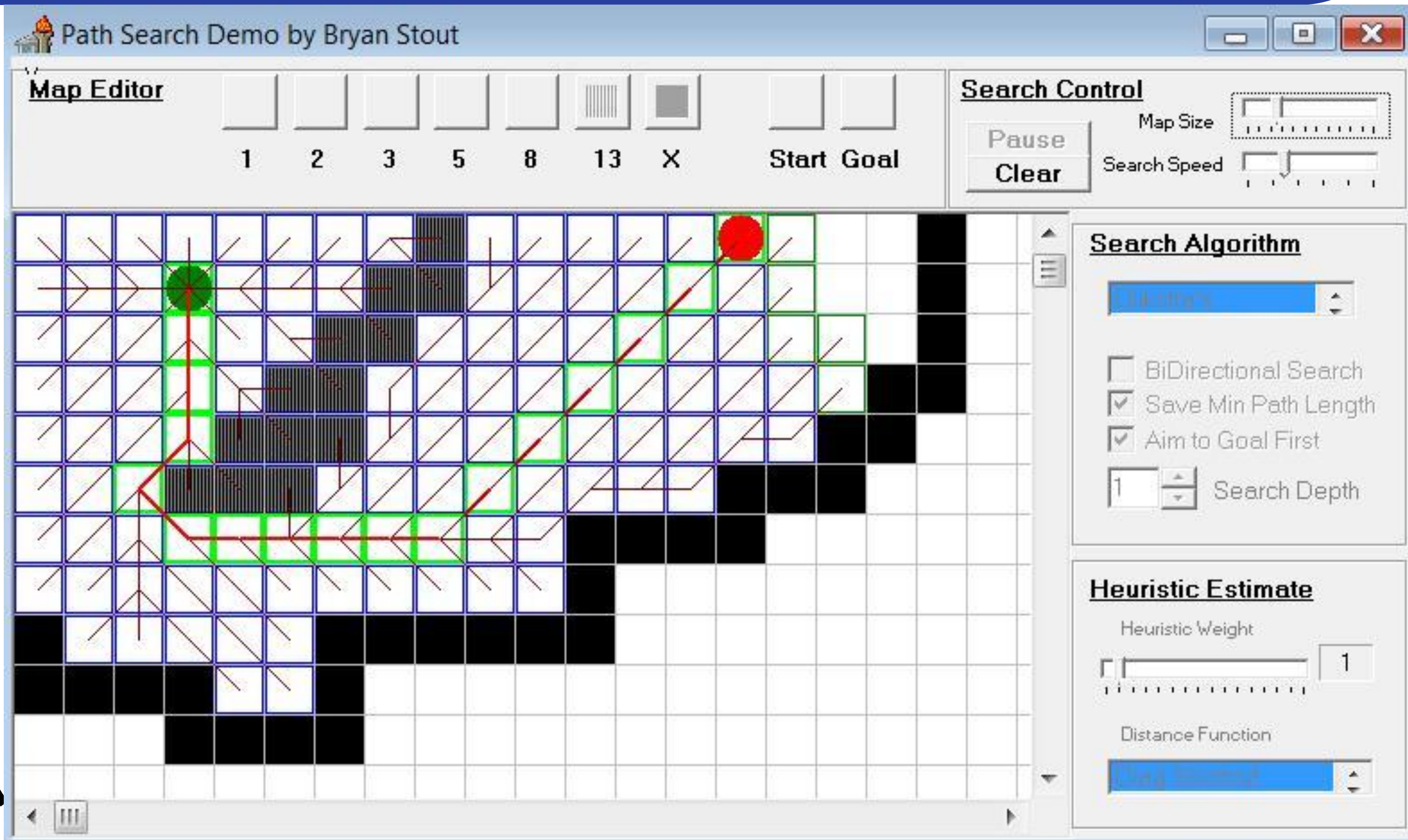
Cuprins

- Explorarea spațiului stărilor problemei
- Explorare informată irevocabilă
- Explorări tentative informate
  - Explorare lacomă
  - Explorare tentativă completă
  - Explorare A*

Probleme cu căutările neinformate

- Modelul unor probleme este prea complicat → variantele de rezolvare se bazează pe explorarea spațiului stărilor.
- Probleme:
  - Deseori se calculează prea mult (ex: drumul optim între 2 puncte folosind Dijkstra) - ex: Dijkstra.
  - În cazul grafurilor infinite sau nedescoperite încă, algoritmii clasici fie sunt ineficienți, fie nu garantează găsirea soluției.
- Soluție:
  - Rezolvarea să nu se mai bazeze numai pe calculele exacte ci și pe experiența anterioară (euristici) → direcționarea căutării.

Exemplu Dijkstra



Explorarea spațiului stărilor problemei

Spațiul stărilor unei probleme

- Definiție: Stare a problemei = abstractizare a unei configurații valide a universului problemei, configurație ce determină univoc comportarea locală a fenomenului descris de problemă.

- Definiție: Spațiul stărilor = graf în care nodurile corespund stărilor problemei, iar arcele desemnează tranzițiile valide între stări.
  - Caracteristică importantă: nu este cunoscut apriori, ci este descoperit pe măsura explorării!
  - Descriere
    - Nodul de start (starea inițială);
    - Funcție de expandare a nodurilor (produce lista nodurilor asociate stărilor valide în care se poate ajunge din starea curentă);
    - Predicat de testare dacă un nod corespunde unei stări soluție.

Obiectivele navigării prin spațiul stărilor

- Cartografierea sistematică a spațiului stărilor.

- Asamblarea soluțiilor parțiale care în final conduc la soluția finală. Această soluție finală poate fi:
  - Identificarea stărilor soluție (poziționarea a n regine pe tabla de șah fără să se atace);
  - Drumul străbătut de la starea inițială spre o stare soluție (acoperirea tablei de șah cu un cal);
  - Strategia de rezolvare = arbore multicăi în care rădăcina este starea inițială, iar frunzele sunt stări soluție. În acest arbore, unele noduri corespund unor evenimente neprevăzute care influențează calea de urmat în rezolvare (identificarea monedei false dintr-un grup de 3 monede).

Căutări informate/neinformate; Algoritmi tentativi/irevocabili

- Definiție: Dacă explorarea se bazează pe informația acumulată în cursul explorării, informație prelucrată euristic (costuri)→ algoritm informat.

- Definiție: Dacă explorarea este 'la întâmplare' → algoritm neinformat.

- Definiție: Dacă algoritmul de explorare are posibilitatea să abandoneze calea curentă de rezolvare și să revină la o cale anterioară → algoritmi tentativi.

- Definiție: Altfel (algoritmul avansează pe o singură direcție) → algoritmi irevocabili.

Căutări informate vs neinformate

- Căutările informate beneficiază de informații suplimentare pe care le colectează și le utilizează în încercarea de a ghici direcția în care trebuie explorat spațiul stărilor pentru a găsi soluția.
- Aceste informații sunt stocate:
  - În nodurile din spațiul stărilor:
    - Starea problemei reprezentată de nod;
    - Părintele nodului curent;
    - Copii nodului curent (obținuți prin expandarea acestuia);
    - Costul asociat nodului curent care estimează calitatea nodului f(n);
    - Adâncimea de explorare.
  - În structuri auxiliare pentru diferențierea nodurilor în raport cu gradul de prelucrare:
    - Expandat (închis) – toți succesorii nodului sunt cunoscuți;
    - Explorat (deschis) – nodul e cunoscut, dar succesorii săi nu;
    - Neexplorat – nodul nu e cunoscut.

Listele CLOSED și OPEN

- OPEN = mulțimea (lista) nodurilor explorate (frontiera dintre zona cunoscută și cea necunoscută).
- CLOSED = mulțimea (lista) nodurilor expandate (regiunea cunoscută în totalitate).
- Explorarea zonelor necunoscute se face prin alegerea și expandarea unui nod din OPEN. După expandare, nodul respectiv e trecut în CLOSED.
- Majoritatea algoritmilor tentativi folosesc lista OPEN, dar doar o parte folosesc lista CLOSED.

Completitudine și optimalitate

- Definiție: Algoritm complet = algoritm de explorare care garantează descoperirea unei soluții, dacă problema acceptă soluție.
  - Algoritmii irevocabili sunt mai rapizi și consumă mai puține resurse decât cei tentativi, dar nu sunt compleți pentru că pierd informație.
- Definiție: Algoritm optimal = algoritm de explorare care descoperă soluția optimă a problemei.

Ø

Algoritm generic de explorare

- Explorare(StInit, test_sol)
  - OPEN = {constr_nod(StInit)}; // starea inițiala
  - **Cât timp** (OPEN ≠ Ø)

    // mai am noduri de prelucrat
    - nod = selecție_nod(OPEN); // aleg un nod
    - **Dacă** (test_sol(nod)) **Întoarce** nod;

      // am găsit o soluție
    - OPEN = OPEN \ {nod} U expandare{nod};

      // extind căutarea
  - **Întoarce** insucces; // nu s-a găsit nicio soluție

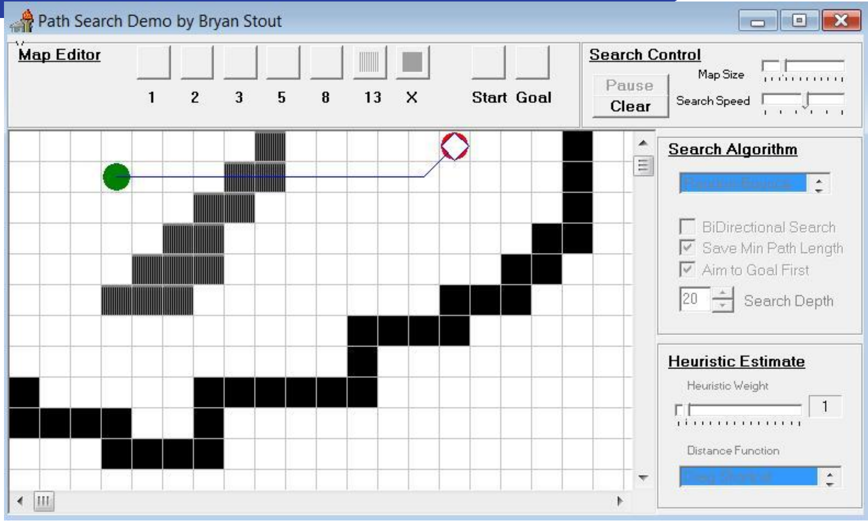
Discuție pe baza algoritmului

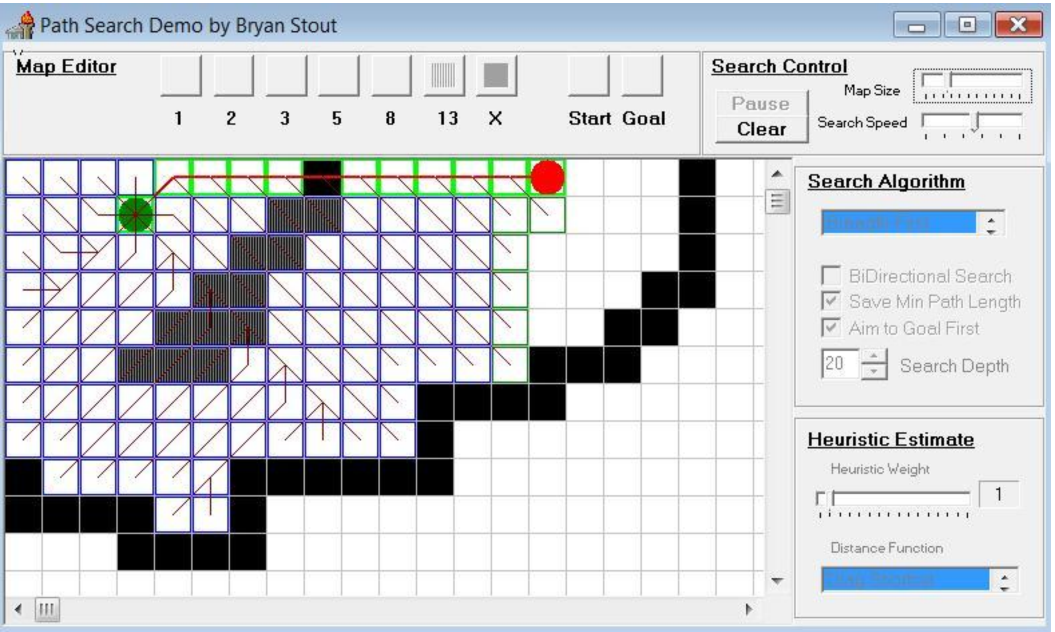
- Dacă **selecție_nod** se realizează independent de costul nodurilor din graful stărilor → căutare neinformată:
  - Dacă e de tip “random” → algoritm aleator - ex: RandomBounce
  - Dacă e de tip “primul venit, primul servit” → OPEN e coadă → BFS – ex: Breadth-first
  - Dacă e de tip “ultimul venit, primul servit” → OPEN e stivă → DFS – ex: Depth-first limitat / IDDFS

- Dacă **selecție_nod** se bazează pe un cost exact sau estimat (euristic) al stărilor problemei → căutare informată:
  - Estimarea costului și folosirea sa în procesul de selecție → esențiale pentru completitudinea, optimalitatea și complexitatea algoritmilor de explorare!

Exemplu de căutări neinformate

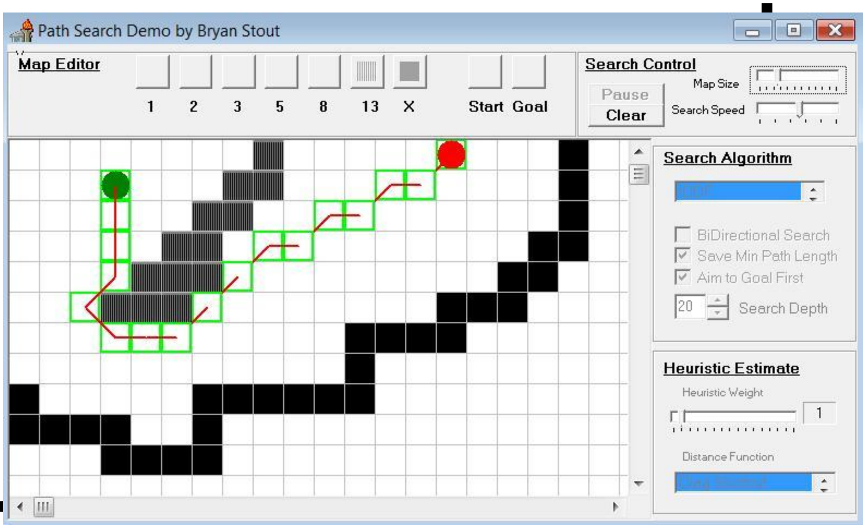
RandomBounce →





BFS↑

IDDFS →



Explorare informată irevocabilă

Algoritm de explorare informată irevocabilă

- Ex: algoritmul alpinistului = algoritmul gradientului maxim.
- Fiecărui nod i se asociază o valoare f(nod) ≥ 0 → calitatea soluției parțiale din care face parte nodul.
- Se păstrează doar cel cu valoare maximă → OPEN are un singur element!

Gradientul Maxim

- Gradient_maxim(StInit, f, test_sol)
  - nod = constr_nod(StInit); // starea inițial
  - π(nod) = null;

Inițializări

  - **Cât timp** (!test_sol(nod))

Testez soluția

    - succs = expandare(nod); // nodurile au o valoare estimata // prin f
    - **Dacă** (succs = Ø) **Întoarce** insucces;

Insucces

// nu mai am noduri de prelucrat

    - succ = selecție_nod(succs); // f(succs) = max {f(n) | n ϵ succs}
    - π(succ) = nod;
    - nod = succ;

Găsesc calea de continuat

  - **Întoarce** nod; // am ajuns la soluție

Soluția

Gradientul Maxim

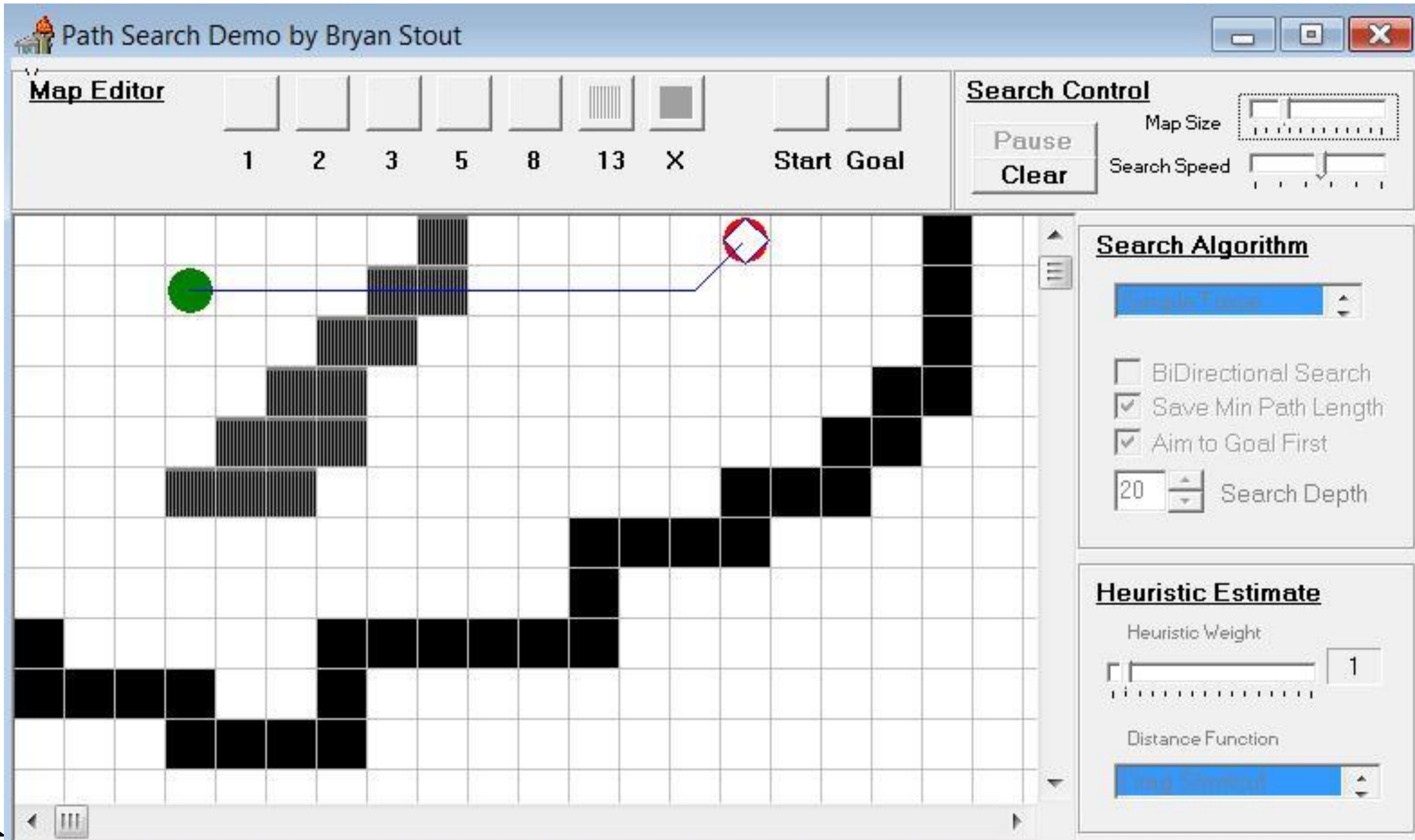
Optimalitate?

Completitudine?

Complexitate?

Ex: SimpleTrace

Exemplu Gradient Maxim



Discuție algoritmul gradientului maxim

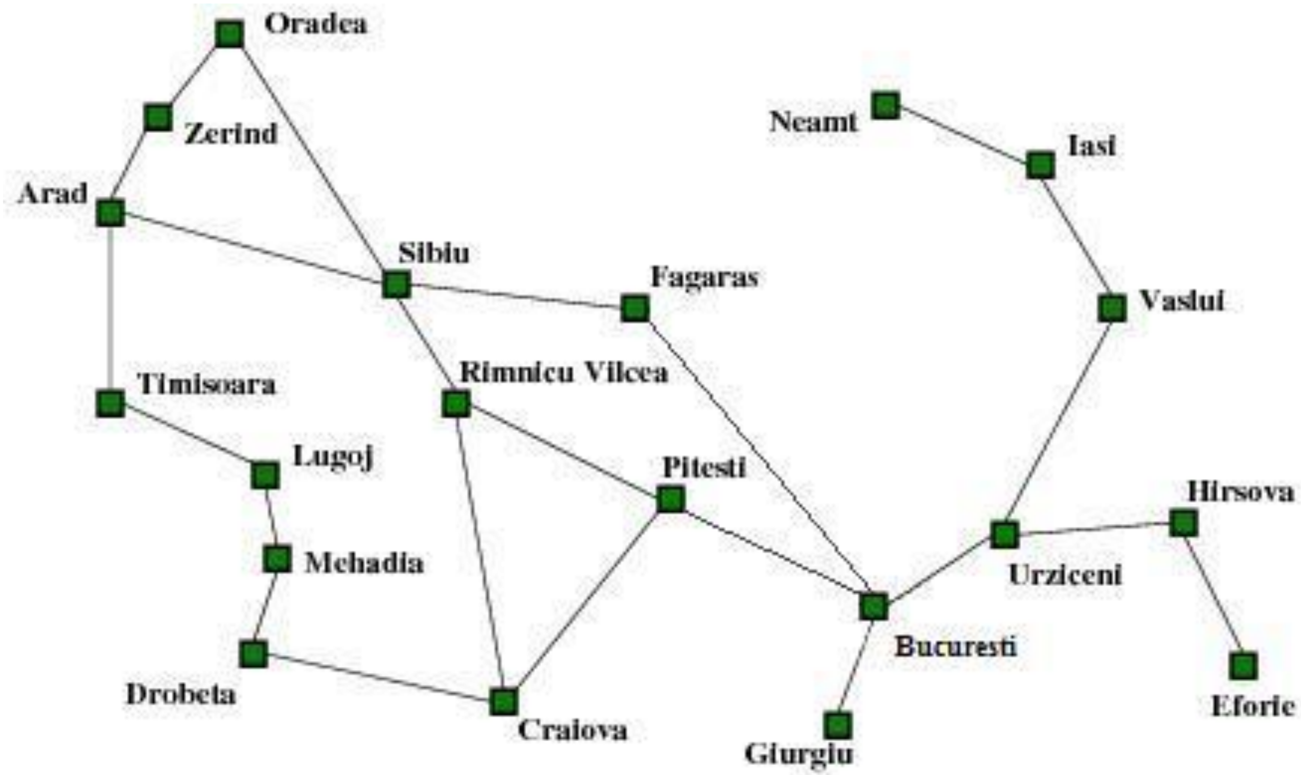
- Algoritmul nu e complet și nu e optimal!
- Complexitate scăzută: O(bd) - b = branching factor, iar d = depth!
- Performanțele algoritmului depind foarte tare de forma teritoriului explorat și de euristica folosită (de dorit să existe puține optime locale și o euristică de evaluare cât mai bună).
- Pseudo-soluție eliminare optim local: se lansează algoritmul de mai multe ori plecând din stări inițiale diferite și se alege cea mai bună soluție obținută.

Explorări tentative informate

Detalii generale

- Păstrează toate nodurile de pe frontieră (OPEN), unii păstrând și nodurile expandate (CLOSED).
- Fiecare nod are un cost asociat $f(n) \geq 0$ care estimează calitatea nodului (distanța de la nodul respectiv până la un nod soluție).
- Cu cât $f(n)$ este mai mic, cu atât nodul este mai bun.

Prezentarea problemei



- Trebuie să ajungem în București din diverse puncte ale țării pe ruta cea mai scurtă.

Explorare lacomă

- Explorare_lacomă (StInit, f, test_sol)
  - nod = constr_nod(StInit); // starea inițiala
  - π(nod) = null;
  - OPEN = {nod};

    Inițializări
  - **Cât timp** (OPEN ≠ Ø) // mai am noduri de prelucrat
    - nod = selecție_nod (OPEN); // f(nod) = min {f(n) | n ∊ OPEN}
    - **Dacă** (test_sol(nod)) **Întoarce** nod;

      Soluția
    - OPEN = OPEN \ {nod}; // nodul nu e soluție, trebuie expandat
    - succs = expand(nod); // expandare nod
    - **Pentru fiecare** (succ ∊ succs) // actualizare succesori
      - OPEN = OPEN U {succ};
      - π(succ) = nod;

      Continuarea căutării
  - **Întoarce** insucces;

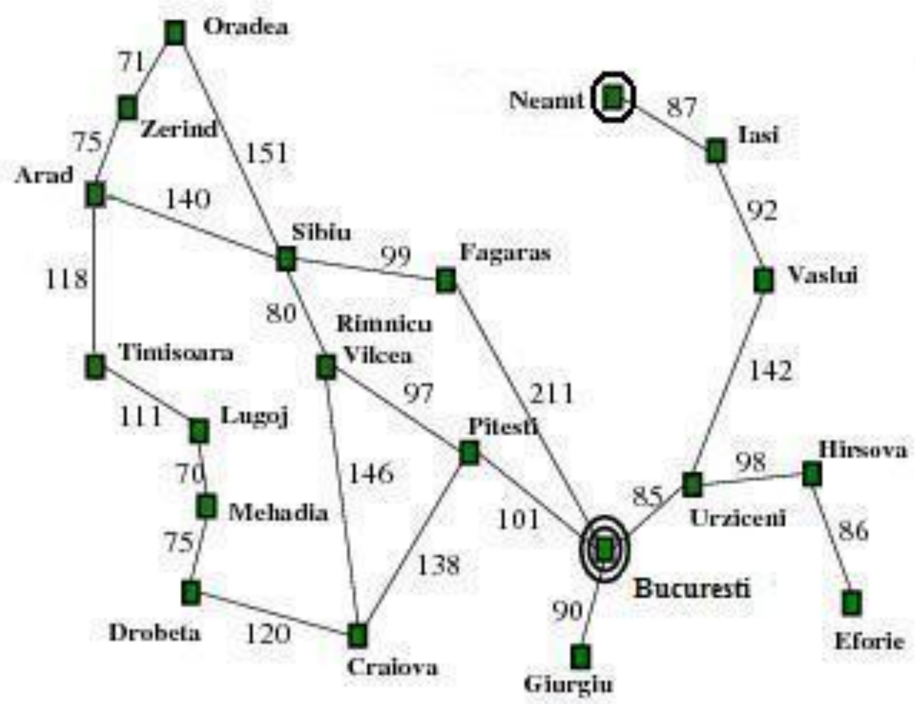
    Insucces

Optimalitate?

Completitudine?

Problema?

f(nod) = distanța de la nodul curent până la nodul nod



f(Iași) = 87 → Iași

f(Neamț) = 87

f(Vaslui) = 92 → Neamț

- Drumul Neamț-București? → nu se termină algoritmul!
- → Explorarea lacomă nu e completă → trebuie să se rețină teritoriul deja parcurs ca să se evite ciclurile!

Explorare tentativă completă BF* (BEST FIRST) (1)

- BF*(StInit, f, test_sol)
  - nod = constr_nod(StInit); // starea inițială
  - π(nod) = null;
  - OPEN = {nod}; // noduri explorate dar neexpandate
  - CLOSED = Ø; // noduri expandate

  Inițializări

  - **Cât timp** (OPEN ≠ Ø)
    - nod = selecție_nod (OPEN); // f(nod) = min {f(n) | n ∊ OPEN}
    - **Dacă** (test_sol(nod)) **Întoarce** nod;

    Soluția

    - OPEN = OPEN \ {nod};
    - CLOSED = CLOSED U {nod};
    - succs = expand(nod);

    Continuarea căutării

Explorare tentativă completă BF* (BEST FIRST) (2)

- **Pentru fiecare** (succ ∊ succs)
  - **Dacă** (succ ∉ CLOSED U OPEN) **atunci**
    - OPEN = OPEN U {succ}; π(succ) = nod;

Nod nou

  - **Altfel**
    - succ' = apariția lui succ în CLOSED U OPEN
    - **Dacă** (f(succ) < f(succ')) // am găsit o cale mai bună către succ și
      // redeschidem nodul

      π(succ') = nod; // actualizez părintele

      f(succ') = f(succ); // și costul nodului

Actualizări

      Dacă (succ' ∊ CLOSED) // dacă era considerat expandat, îl redeschid

      CLOSED = CLOSED \ {succ'}; OPEN = OPEN U {succ'};

Reprelucrare

- **Întoarce** insucces;

Insucces

Optimalitate?

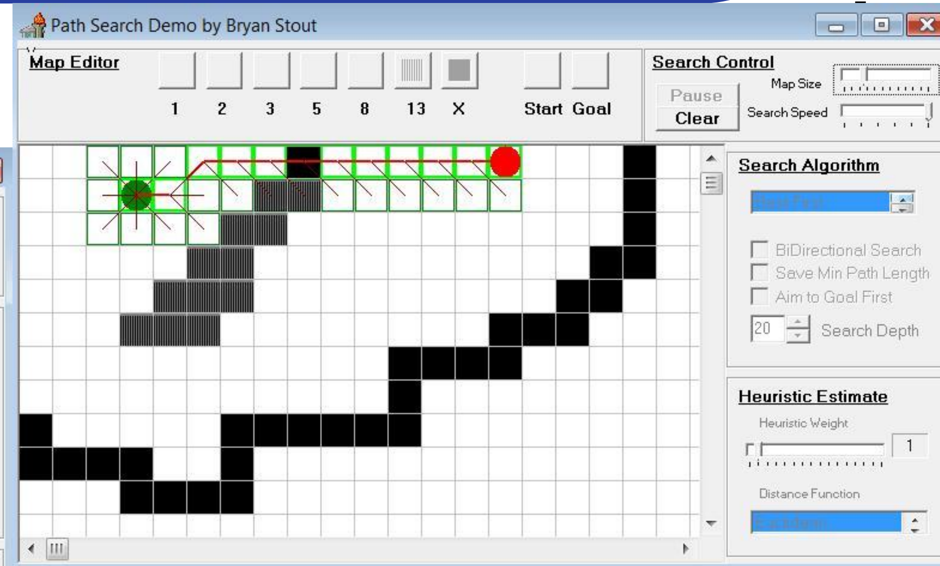
Completitudine?

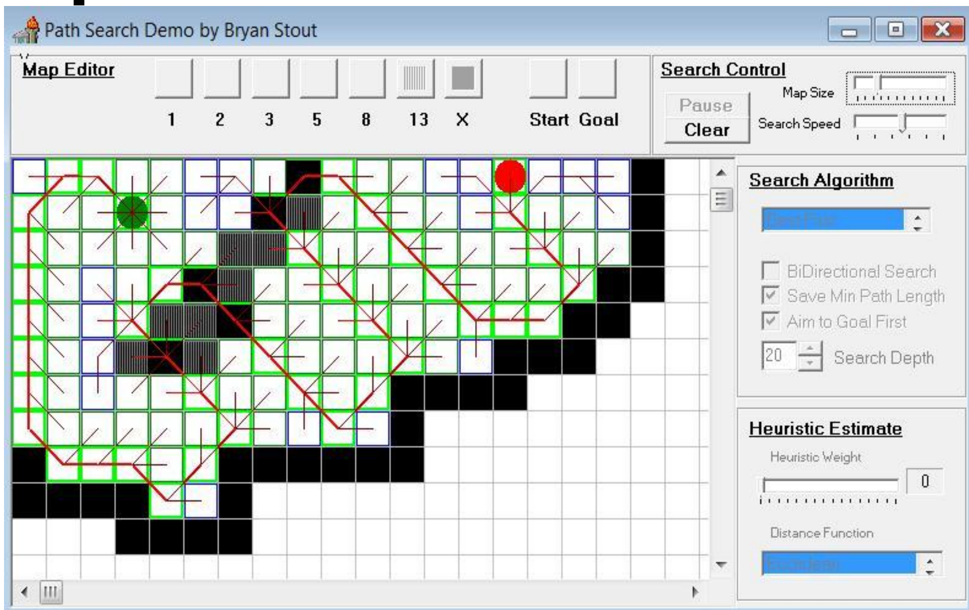
Complexitate?

ex: Best-first cu diverse euristici

Exemple rulare BF* cu diferite euristici

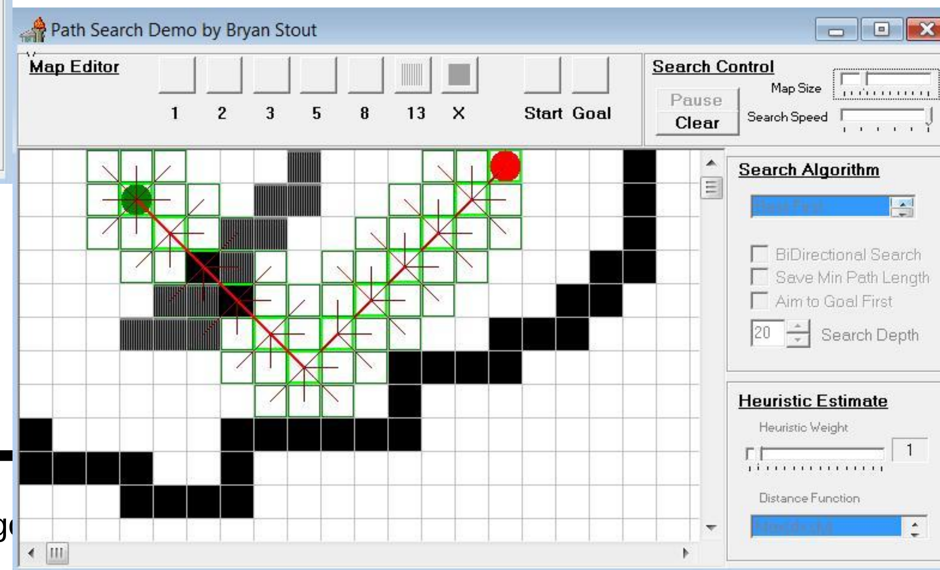
BF* - Euristică Distanta Euclidiană (11 pași, cost 23) →





BF* fără euristici ↑

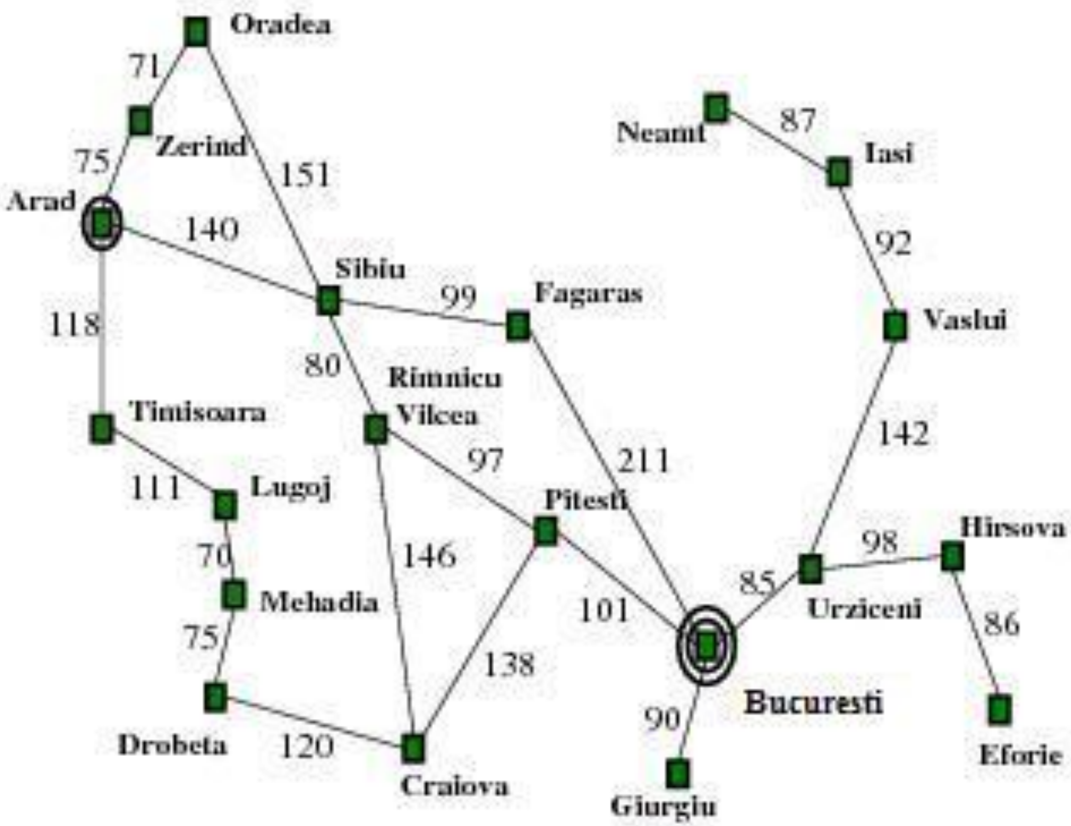
BF* - Euristică maximul distantei pe x și pe y (11 pași, cost 35)→



BF* - completitudine, optimalitate și complexitate

- Păstrează întreg teritoriul explorat:
  - OPEN – nodurile de pe frontieră;
  - CLOSED – nodurile expandate (unele noduri pot fi redeschise) → se evită ciclurile.
- Algoritmul este complet dar nu este optim → optimalitatea depinde de euristica f!
- Complexitate: $O(b^{d+1})$

Aplicație BF*



Distanta in linie dreapta pana la Bucuresti	
Arad	366
Craiova	160
Drobeta	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

- Drumul optim Arad-București (f(nod) = distanța în linie dreaptă până la București)

A*

- Variantă a BF*.
- Nu poate fi aplicat mereu → trebuie demonstrat că păstrează ordinea soluțiilor unde soluțiile problemelor sunt drumuri în spațiul stărilor! (vezi Giumale pentru detalii!)
- Costul unui drum este aditiv (= suma costurilor arcelor) și crescător în lungul drumului.
- Folosește două funcții de cost:
  - h(n) - distanța estimată de la nodul curent până la nodul țintă;
  - g(n) - distanța parcursă de la nodul inițial până la nodul curent;
  - f(n) = g(n) + h(n).

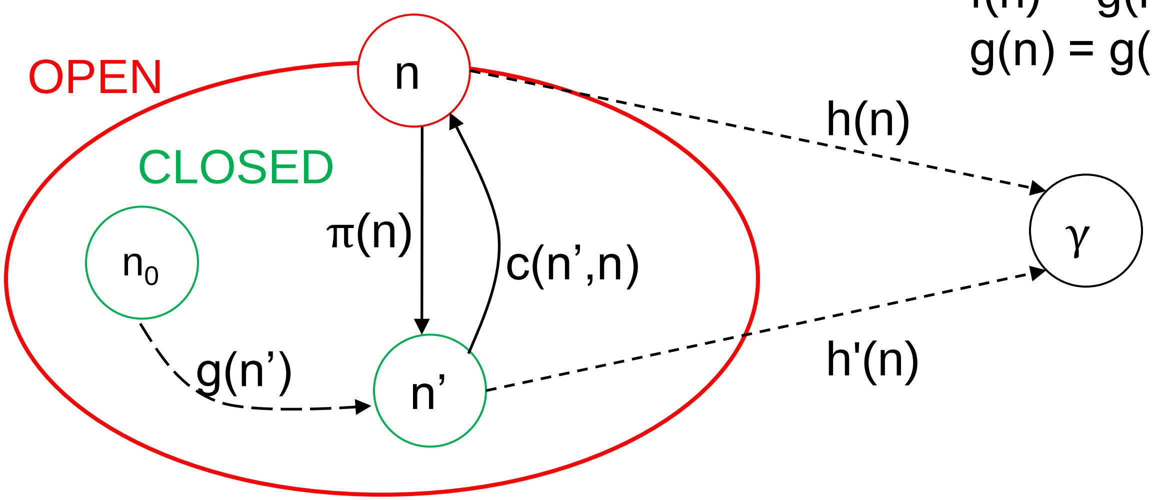
Notații (1)

- **S = (V,E)** – graful asociat spațiului stărilor problemei;
- **n_0** – nodul de start asociat stării inițiale a problemei;
- **$\Gamma \subseteq V$** – mulțimea nodurilor soluție. Un nod soluție se notează γ;
- **c(n,n') > 0** – costul arcului (n,n');
- **π(n)** – părintele lui n;
- **g(n)** – costul drumului n_0..n descoperit de algoritm la momentul curent de timp;
- **g_p(n)** – costul exact al porțiunii n_0..n din lungul unei căi date P;
- **g*(n)** – costul exact al unui drum optim n_0..n;

Notații (2)

- **$h(n) \geq 0$** – costul estimat al drumului optim de la nodul n la cel mai favorabil nod soluție $\gamma \in \Gamma$. În plus $h(\gamma) = 0$, pentru orice $\gamma \in \Gamma$;
- **$h^*(n)$** – costul exact al porțiunii de drum optim $n..\,\gamma$, pentru cel mai favorabil nod $\gamma \in \Gamma$ ($h^*(n) = \min \{cost(n..\,\gamma) \mid \gamma \in \Gamma\}$);
- **$f(n) = g(n) + h(n)$** – costul estimat al întregului drum $n_0..n..\,\gamma$, pentru cel mai favorabil nod $\gamma \in \Gamma$, unde porțiunea de drum $n_0..n$ este cea descoperită de algoritm la momentul curent de timp în cursul execuției;
- **$f^*(n) = g^*(n) + h^*(n)$** – costul exact al unui drum optim $n_0..n..\,\gamma$, pentru cel mai favorabil nod $\gamma \in \Gamma$;
- **$C = \min\{f^*(\gamma) \mid \gamma \in \Gamma\}$** – costul exact al unui drum optim $n_0..\,\gamma$, $\gamma \in \Gamma$. (C = costul soluției optime);

Funcția de evaluare A*



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

$g(n) = g(n') + c(n',n)$

A* (1)

- A*(StInit, h, test_sol)
  - n_0 = constr_nod(StInit); // starea inițială

    Inițializări

  - $f(n_0) = h(n_0)$; $g(n_0) = 0$; $\pi(n_0)$ = null; // euristici
  - OPEN = $\{n_0\}$; CLOSED = Ø; // și cozi
  - **Cât timp** (OPEN ≠ Ø) // mai am noduri de prelucrat
    - nod = selecție_nod (OPEN); // f(nod) = min {f(n) | n ∊ OPEN}
    - **Dacă** (test_sol(nod)) **Întoarce** nod;

      Soluția

    - OPEN = OPEN \ {nod}; // updatez OPEN
    - CLOSED = CLOSED U {nod}; // și CLOSE
    - succs = expand(nod); // determin nodurile succesoare

      Continuarea căutării

A* (2)

- **Pentru fiecare** (succ ∊ succs) { // prelucrare succs
  - g_succ = g(nod) + c(nod, succ); // calculez g
  - f_succ = g_succ + h(succ); // calculez f = g + h

Prelucrare succesori

- **Dacă** (succ ∉ CLOSED U OPEN) **atunci** // nod nou descoperit →
  - OPEN = OPEN U {succ}; // îl bag în OPEN
    g(succ)= g_succ; f(succ)= f_succ; π(succ) = nod;

Nod nou

- **altfel** // a mai fost prelucrat
  - **Dacă** (g_succ < g(succ)) // verific dacă noul g este mai mic decât // anteriorul
    g(succ)= g_succ; f(succ)= f_succ; π(succ) = nod; // cale mai bună

Actualizări

Reprelucrare

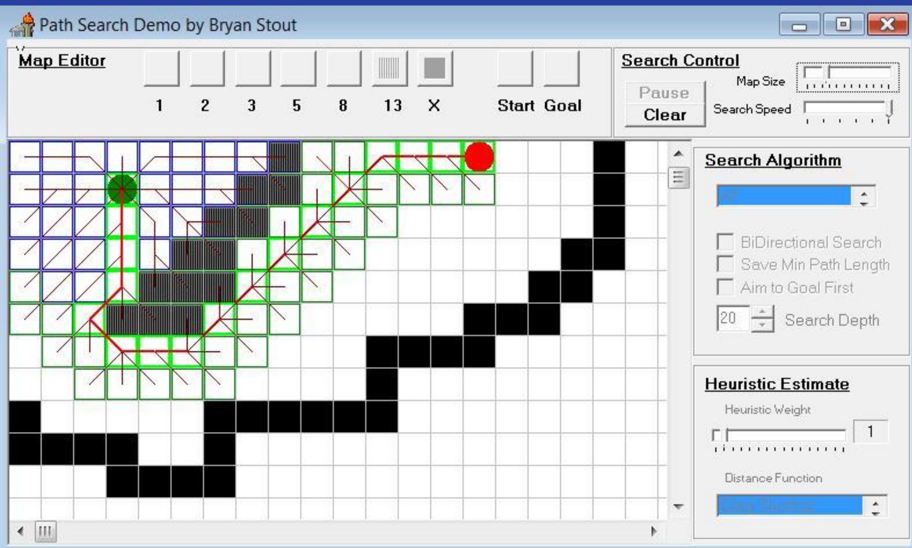
Dacă (succ ∊ CLOSED) // dacă era considerat expandat, îl redeschid
CLOSED = CLOSED \ {succ'}; OPEN = OPEN U {succ'};

- **Întoarce** Insucces;

Insucces

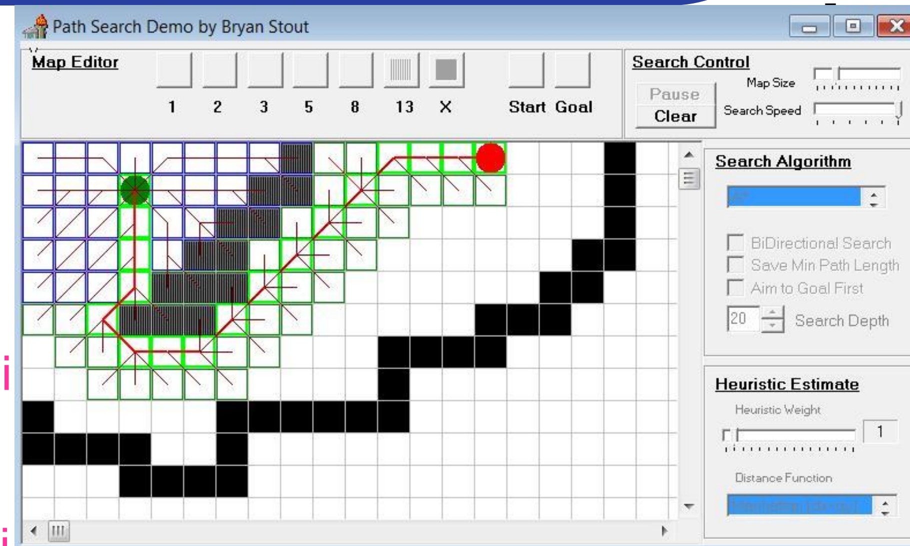
ex: A* cu diverse euristici

Exemple A* cu diverse euristici



A*

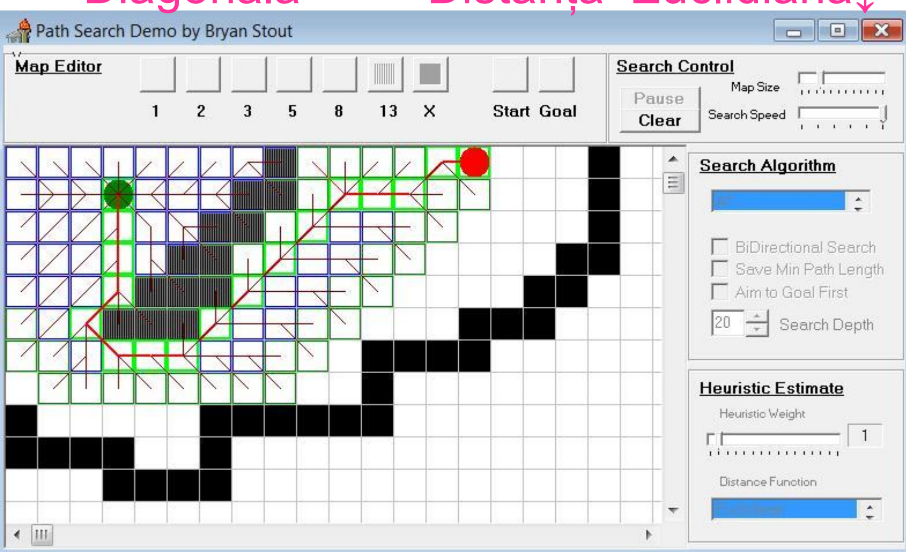
16 pași



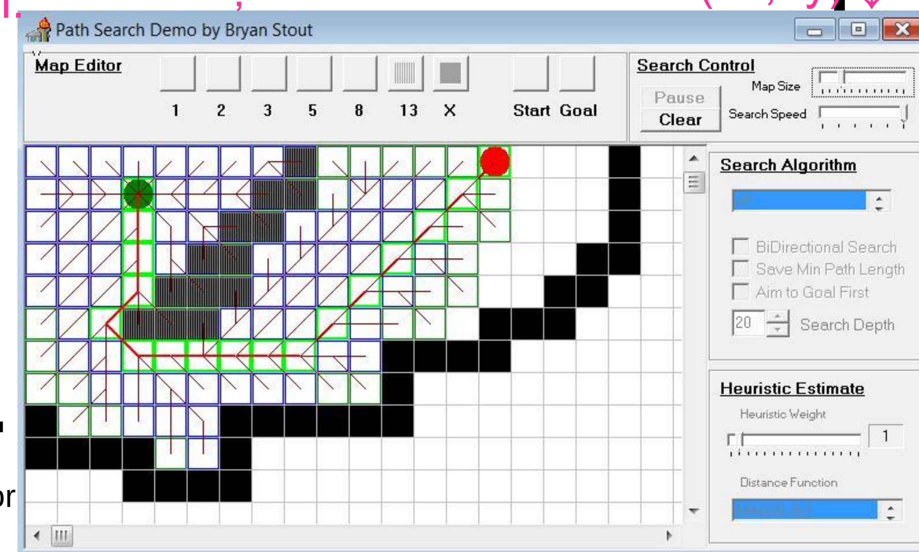
Diagonală↑ Distanța Euclidiană↓

Euristici:

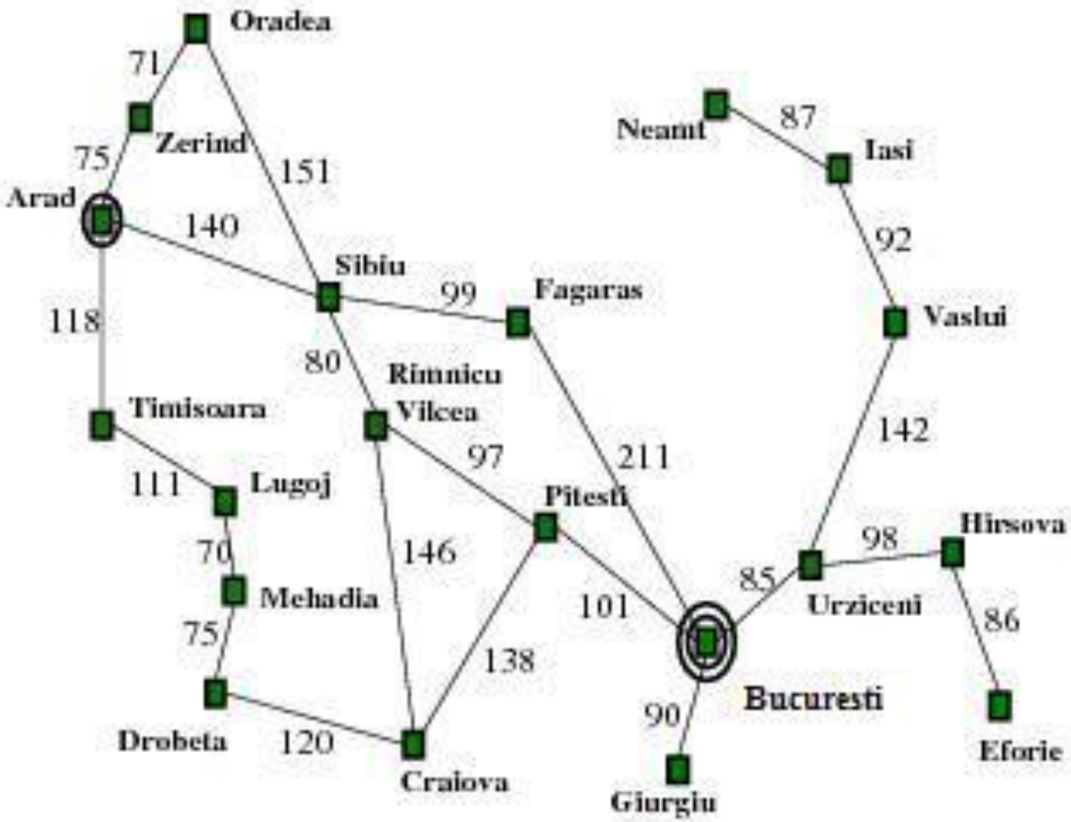
Distanța Manhattan ↑ Max(dx,dy) ↓



a Algor



Aplicație A*

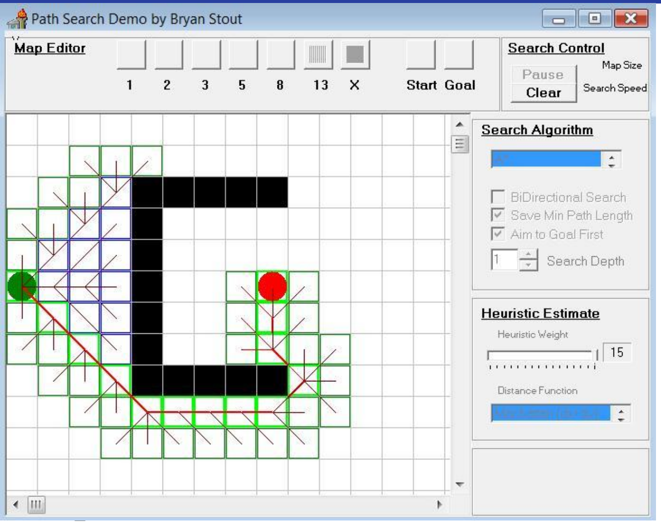


Distanta in linie dreapta pana la Bucuresti

Arad	366
Craiova	160
Drobeta	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

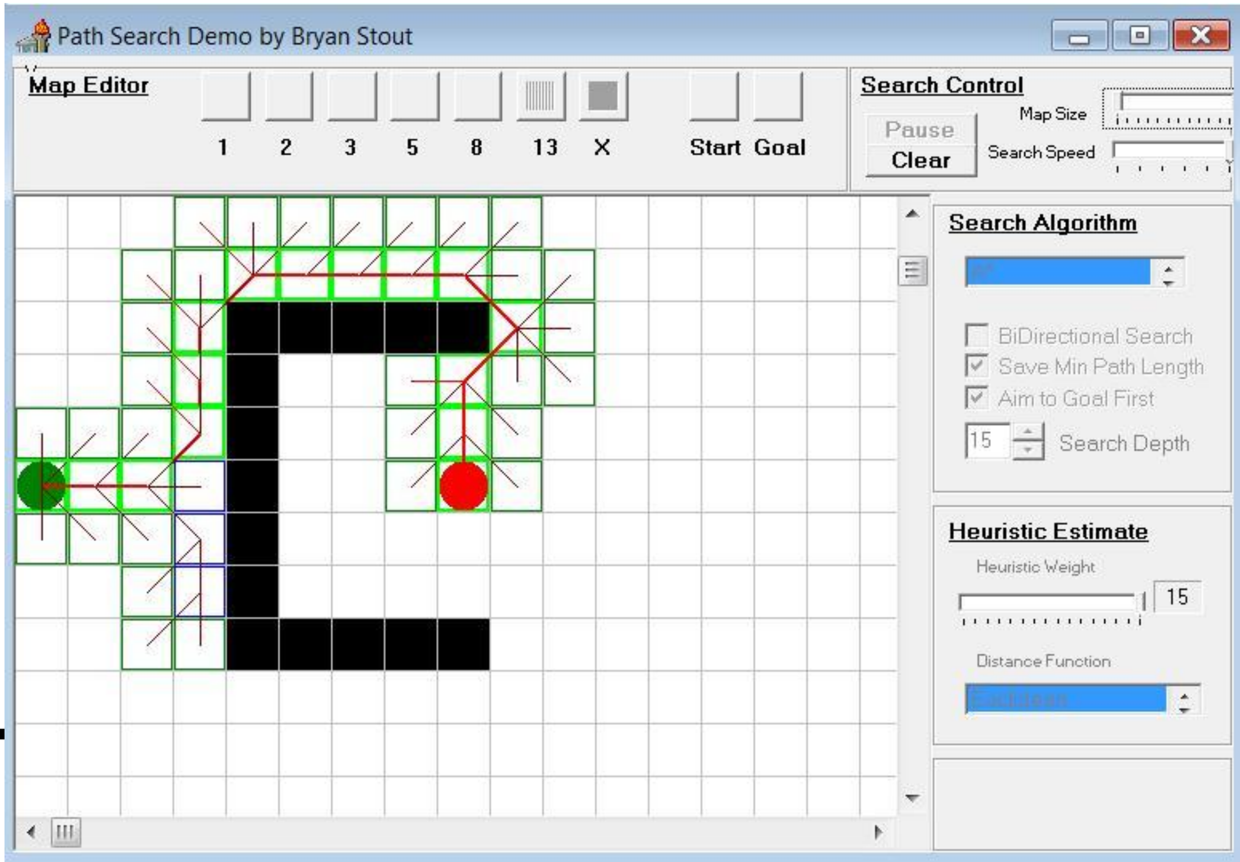
- Drumul optim Arad-București (h(n) = distanța în linie dreaptă până la București, g(n) = distanța parcursă)

Problemă



← A* – Distanța Manhattan – 12 pași

↓ A* – Distanța Euclidiana – 14 pași



Cum se explică???

Algoritmul A* - completitudine și optimalitate (1)

- Teorema 7.1: Algoritmul A* este complet chiar dacă graful explorat nu este finit.

- Lema 7.1: Fie P = $n_0,n_1,\ldots,n_m$ un drum oarecare în graful explorat de A*, astfel încât la un moment T al explorării toate nodurile din P sunt în CLOSED. Atunci, la orice moment de timp egal sau superior lui T, există inegalitatea $g(n_i) \leq g_p(n_i)$, i = 0,m:
  - Costul nodurilor din CLOSED poate să scadă, dar de fiecare dată când acest lucru se întâmplă, se pierde timp → scoaterea nodului din CLOSED, punerea în OPEN, prelucrarea acestuia încă o dată → trebuiesc evitate aceste situații → alegerea unei euristici cât mai bune care să minimizeze numărul acestor actualizări!

Algoritmul A* - completitudine și optimalitate (2)

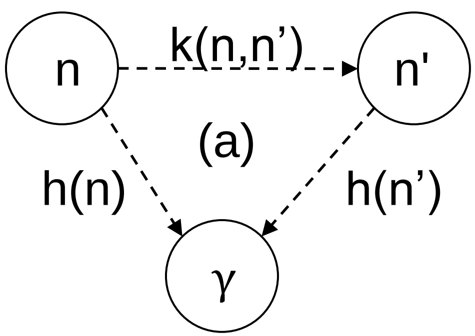
- Definiție 7.2: Funcția euristică h este admisibilă dacă pentru orice nod n din spațiul stărilor $h(n) \leq h^*(n)$. Cu alte cuvinte, o euristică admisibilă h este optimistă și $h(\gamma) = 0$ pentru orice nod $\gamma \in \Gamma$.

- Teorema 7.2: Algoritmul A* ghidat printr-o euristică admisibilă descoperă soluția optimă dacă există soluții.

Euristici – consistență și monotonie

- Definiție 7.4: O euristică h este consistentă dacă pentru oricare două noduri n și n’ ale grafului explorat, astfel încât n’ este accesibil din n, există inegalitatea: $h(n) \leq h(n') + k(n,n')$, unde $k(n,n')$ este costul unui drum optim de la n la n’.

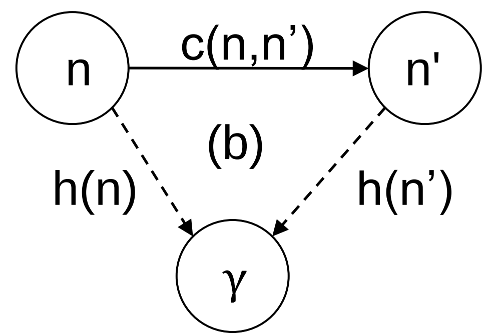
- Definiție 7.5: O euristică h este monotonă dacă pentru oricare două noduri n și n’ ale grafului explorat, astfel încât n’ este succesorul lui n, există inegalitatea $h(n) \leq h(n') + c(n,n')$, unde $c(n,n')$ este costul arcului (n,n’).



$h(n) \leq h(n') + k(n,n')$

Regula triunghiului pentru euristici:

← Consistență
→ Monotone



$h(n) \leq h(n') + c(n,n')$

Consistență = monotonie

- Teorema 7.5: O euristică este consistentă ⇔ este monotonă.
  - Demonstrație:
    - h – consistentă → h – monotonă. Alegem n' ∈ succs(n) → $k(n,n') = c(n,n')$ → $h(n) \le h(n') + c(n,n')$ → h – monotonă.
    - h – monotonă → h – consistentă. Fie $n = n_1, n_2, \ldots, n_q = n'$, un drum optim n..n' cu cost k(n,n'). → $h(n) = h(n_1) \le h(n_2) + c(n_1,n_2) \le h(n_3) + c(n_1,n_2) + c(n_2,n_3) \ldots \le h(n_q) + c(n_1,n_2) + c(n_2,n_3) + \ldots c(n_{q-1},n_q) = h(n_q) + k(n_1,n_q)$ → $h(n) \le h(n') + k(n,n')$ → h – consistentă.

Consistență → admisibilitate

- Teorema 7.6: O euristică consistentă este admisibilă.
  - Demonstrație:
    - Fie h o euristică consistentă → $h(n) \le h(n') + k(n,n')$, $\forall n'$ accesibil din n. Fie $n' = \gamma \in \Gamma$ → $k(n, \gamma) = \min \{ k(n, \gamma') \mid \gamma' \in \Gamma\} = h^*(n)$ → $h(n) \le h(\gamma) + h^*(n)$, dar $h(\gamma) = 0$ → $h(n) \le h^*(n)$ → euristică admisibilă.
- Corolar 7.2: O euristică monotonă este admisibilă.

Dominanță - Definiții și teoremă

- Definiție 7.6: Fie h_1 și h_2 două euristici admisibile. Spunem că h_1 este mai informată decât h_2 dacă $h_2(n) < h_1(n)$ pentru orice nod $n \notin \Gamma$ din graful spațiului de stare explorat.

- Definiție 7.7: Un algoritm A_1* domină un algoritm A_2* dacă orice nod expandat de A_1* este expandat și de A_2*. (eventual, A_2* expandează noduri suplimentare față de A_1*, deci A_1* poate fi mai rapid ca A_2*.)

- Teorema 7.11: Dacă o euristică monotonă h_1 este mai informată decât o euristică monotonă h_2, atunci un algoritm A_1* condus de h_1 domină un algoritm A_2* condus de h_2.

Dominanța - Exemplu

- Considerăm jocul 8-pătrățele care trebuie aranjat pornind de la forma inițială prin mutarea locului 'liber' astfel încât să ajungem la forma finală:

7	4	1
5	6	3
2	8	

→

1	2	3
4		5
6	7	8

- Două euristici posibile:
  - h_1 = numărul pătrățelelor a căror poziție curentă diferă de poziția finală;
  - $h_1 = \Sigma_{p \in piese}(\delta_p)$, unde $\delta_p = 0$ dacă poziția curentă coincide cu cea finală și $\delta_p = 1$, altfel
  - h_2 = distanța Manhattan = suma distanțelor pe verticală și orizontală între pozițiile curente ale pătrățelelor și pozițiile lor finale
  - $h_2 = \Sigma_{p \in piese}(dist\_h_p + dist\_v_p)$

Admisibilitate? Monotonie? Dominanță? Care euristică va fi aleasă pentru A*?

Complexitate A*

- Liniară dacă $|h^*(n) - h(n)| \leq \delta$, unde $\delta \geq 0$ este o constantă.
- Subexponențială, dacă $|h^*(n) - h(n)| \leq O(\log(h^*(n)))$.
- Exponențială, altfel, (dar mult mai bună decât a căutărilor neinformate).
- Mai multe explicații găsiți în Giumale 7.4.4!

ÎNTREBĂRI?

Bibliografie

[1] C. Giumale – Introducere in Analiza Algoritmilor – cap. 6.1

[2] Cormen – Introducere in algoritmi – cap. 8.3

[3] http://www.soe.ucsc.edu/classes/cmps102/Spring04/TantaloAsymp.pdf

[4] http://www.mersenne.org/