Spazi di Stati e Ricerca di Soluzioni

Nicola Fanizzi
Ingegneria della Conoscenza
CdL in Informatica • *Dipartimento di Informatica*
Università degli studi di Bari Aldo Moro

indice

Risoluzione di Problemi mediante Ricerca

Problem Solving tramite Ricerca

Caso semplice: il sistema ragiona, in *assenza di incertezza*, su un *modello* del mondo fatto di *stati* con un *obiettivo* da raggiungere:

- rappresentazione *piatta* (non gerarchica)
- nello spazio degli stati,
  si deve andare dallo stato corrente a un *obiettivo* *goal*
  - trovare la sequenza di passi *prima* di agire per raggiungerlo

Astrazione del problema: ricerca di un *percorso* in un *grafo orientato* da un nodo di partenza a un nodo finale

- molti algoritmi disponibili
- computazione/ricerca nella *rappresentazione interna* del sistema

Esempio — *Navigatore*

- ricerca del *miglior* percorso
  - più corto
  - più veloce
  - di costo minimo (carburante, tempo, ecc.)
- stato = localizzazione corrente (+ direzione + velocità)

Idea — il sistema costruisce una serie di *soluzioni parziali* (percorsi)

- *verifica* sulle soluzioni parziali
  - soluzione del problema?
  - può portare alla soluzione del problema?
- *ricerca* come ripetizione di:
  - selezione di una soluzione parziale
  - `se` questa costituisce un percorso verso uno dei goal: `stop` `altrimenti` costruire nuove soluzioni parziali
    - estendendo quella corrente (nei doversi modi possibili)

Ricerca strategia comune a molti problemi in AI:

- il sistema riceve solo una descrizione di *COSA* rappresenti una soluzione
  - *COME* ottenerla (algoritmo) → va cercata
- modalità di verifica *efficiente* del raggiungimento di goal
  - anche con algoritmi / problemi NP-completi

Spesso problemi difficili anche per gli uomini

- senza struttura mappabile a caratteristiche del mondo fisico
  - ad es. in *crittografia*: problemi difficili anche per le macchine date limitazioni di tempo e spazio
- ma si può sfruttare conoscenza aggiuntiva (*euristica*) nella ricerca

Spazi di Stati

Formulazione problema in termini di uno **spazio di stati**

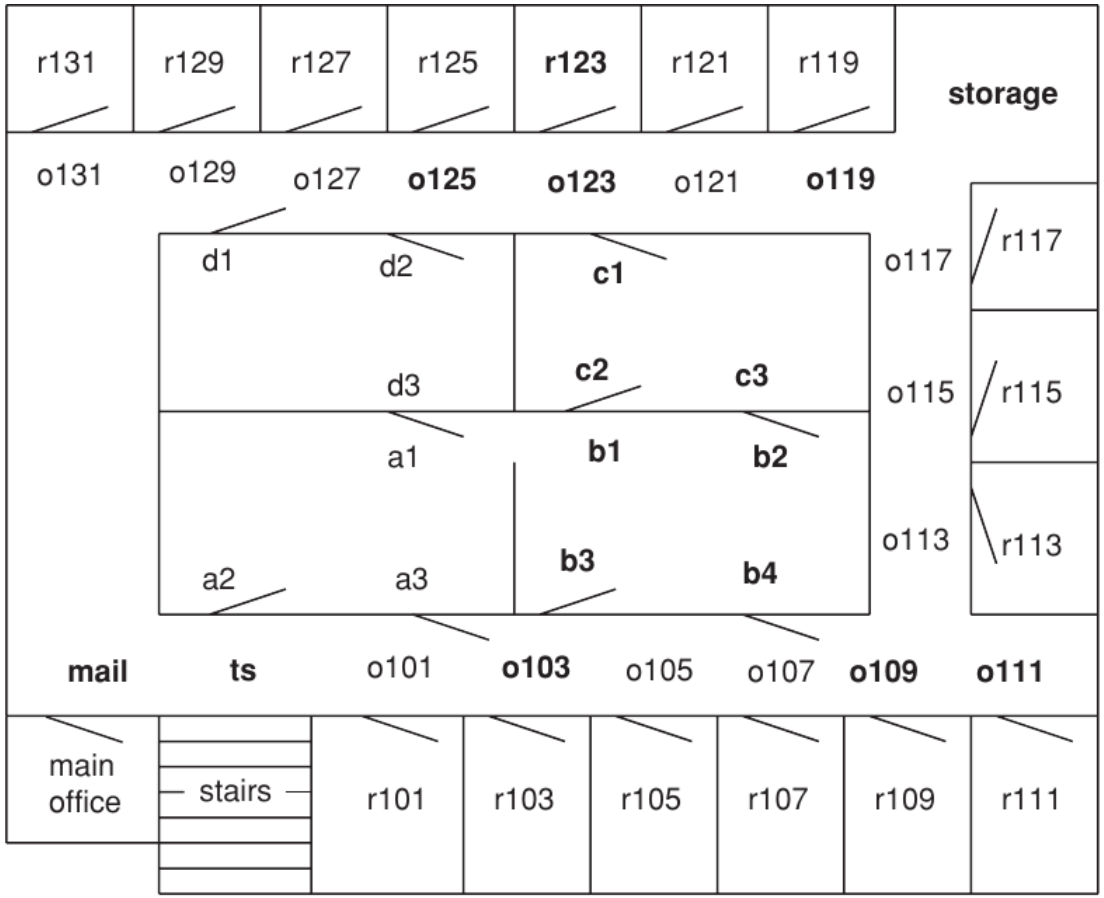
- uno **stato** contiene tutta l'informazione necessaria a predire gli effetti di un'azione e a verificare se si tratti di un goal
- **soluzione**: sequenza di azioni che portano dallo stato corrente a un goal
  - fine-percorso → soluzione

Assunzioni

- conoscenza *perfetta* dello spazio di ricerca
  - stato corrente *completamente osservabile*
  - presenza di stati-obiettivo: *goal* da raggiungere riconoscibili
- a disposizione, insieme di *azioni* dagli effetti *deterministici* noti

Esempio — robot per le consegne

problema: ricerca di un percorso da un posto ad un altro



- *stati*: posizioni distinte (stanze/esterni-stanza)
- *azioni*: spostamenti da un luogo a un altro nelle vicinanze
- *problema*: portarsi da `o103` a `r123`
- *soluzione*: sequenza di spostamenti

Esempio — robot per le consegne

problema: diverse consegne da fare in posti diversi

- *stato*: posizione robot + consegne in carico al robot + posizione consegne effettuate
- *azioni*: spostamenti, prelievo/rilascio pacchi
- *goal*: le consegne sono nelle locazioni desiderate
  - più stati-soluzione
    - ad es. perché indipendenti dalla pos. finale del robot

Molti dettagli considerati per ora non rilevanti:

- come vengono tenuti i pacchi da parte del robot
- la carica delle batterie
- la fragilità di alcuni pacchi
- ...

Esempio — sistema di tutoring

- *stato*: insieme di argomenti conosciuti da parte dello studente
- *azione*: insegnare una lezione
  - risultato: lo studente apprenderà l'argomento
    - conoscendo anche le propedeuticità
- *obiettivo*: lo studente conosce un certo insieme di nuovi argomenti
  - si potrebbe comprendere nello stato anche il suo grado di *attitudine*

Problemi di Ricerca

Un **problema** di ricerca include:

- *spazio di stati*: insieme di *stati* con
  - sotto-insieme distinto di **stati di partenza**
  - insieme di **stati-obiettivo**
    - specificabili anche attraverso una funzione booleana, $goal(s)$
      - vera sse s è uno stato obiettivo
- insieme di **azioni** per ogni stato
- **funzione d'azione**: (stato, azione) → nuovo stato
- **criterio** di *qualità* per soluzioni *accettabili*
  da cui: **soluzione ottimale** se massimizza il criterio
  - es. *qualsiasi* sequenza di azioni che porti a uno stato-obiettivo
  - es. se ci sono *costi* associati alle azioni: *minimalità del costo totale*
    - a volte soddisfacenti anche soluzioni sub-ottimali
      - e.g. costo = +10% rispetto a quello di una ottimale

Possibili Estensioni

- sfruttamento di *struttura interna* agli stati
- stato *non* completamente *osservabile*
  - es. il robot non conosce la posizione iniziale dei pacchi,
  - es. l'insegnante può non conoscere bene le attitudini di uno studente
- azioni *stocastiche*
  - ad es., studente che può non aver appreso un argomento insegnato
- non stati finali ma *preferenze aggiuntive* complesse
  - in termini di *ricompense* o *punizioni*

Ricerca su Grafo

Ricerca su Grafo

Per risolvere il problema:

- si definisce lo spazio di ricerca
- si applica un algoritmo di ricerca

Molti compiti riconducibili al problema di trovare percorsi in un grafo

- modello *astratto* della soluzione *indipendente* dal particolare dominio

In un *grafo* (*orientato*) fatto di nodi connessi da archi

- trovare un *percorso* (di più archi) tra un nodo di partenza e uno obiettivo

ci possono essere più maniere per rappresentare il problema

Formalizzazione

Un **grafo orientato** consiste di:

- insieme dei *nodi* N
- insieme degli *archi* A
  - coppie *ordinate* nodi

Osservazioni

- potenzialmente insiemi *infiniti*
- rappresentazione *esplicita* o *implicita*
  - mediante procedura per generare nodi e archi all'occorrenza

Nomenclatura

- $\langle n_1, n_2 \rangle$ è **uscente** da n_1 ed **entrante** in n_2
- n_2 è **vicino** di n_1 sse c'è un arco da n_1 a n_2 ossia se $\langle n_1, n_2 \rangle \in A$
  - relazione *non* necessariamente *simmetrica*
  - archi etichettabili con l'*azione* che porta da n_1 a n_2
- **percorso** (o *cammino*, *path*) dal nodo s al nodo g: *sequenza di nodi*

$$\langle n_0, n_1, \ldots, n_k \rangle$$

tale che $s = n_0, g = n_k$ e $\forall i: \ \langle n_{i-1}, n_i \rangle \in A$

  - in alternativa, *sequenza di archi*,

$$\langle n_0, n_1 \rangle, \langle n_1, n_2 \rangle, \ldots, \langle n_{k-1}, n_k \rangle$$

  - o anche *sequenza di etichette* su tali archi

SOLUZIONI

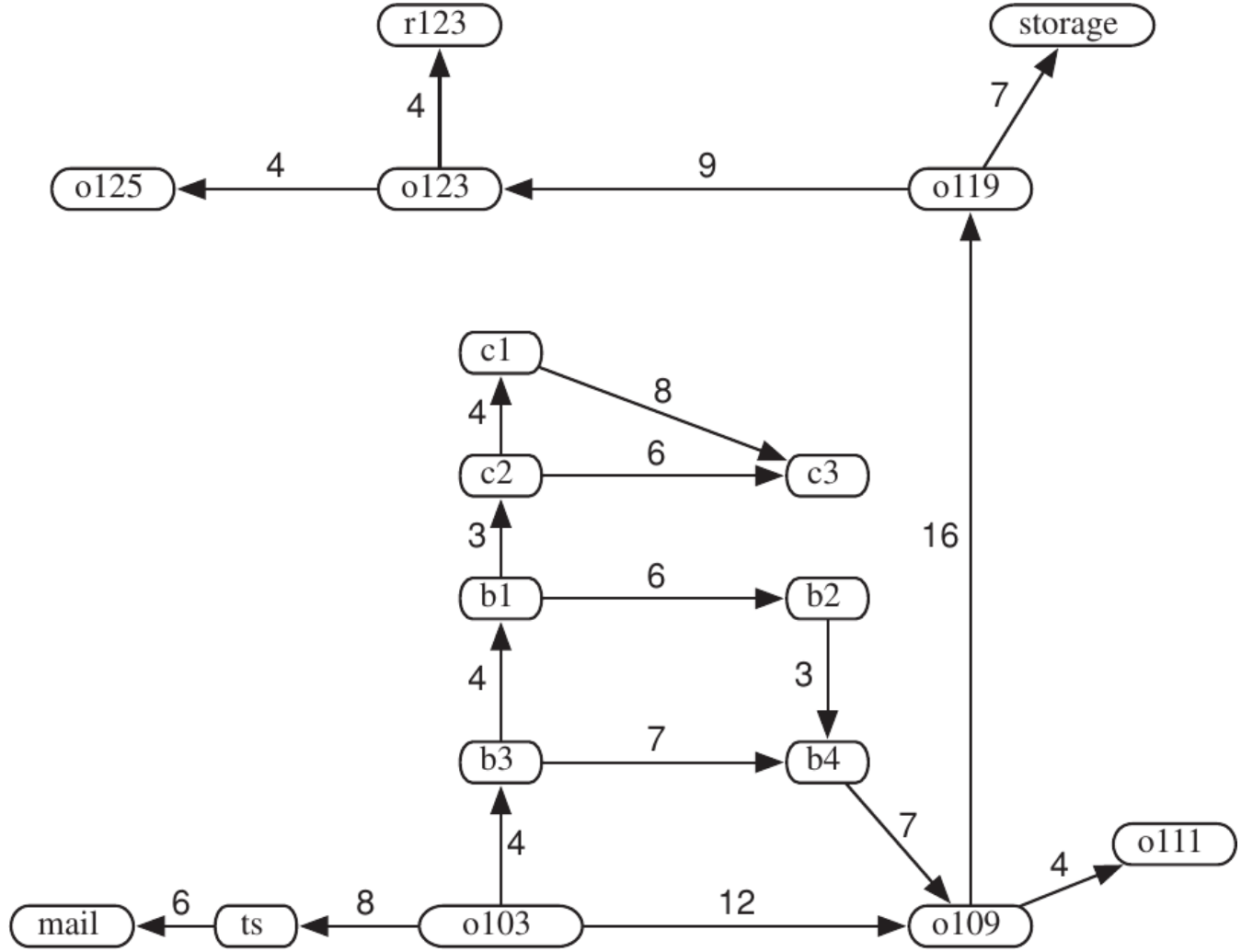
Nel grafo:

- insieme di nodi *di partenza* (*start*)
- insieme dei *nodi-obiettivo* (*goal*)
  - identificabili anche tramite predicato $goal(\cdot)$
- *soluzione*: percorso da un nodo di partenza a uno obiettivo
  - a volte si ammette un **costo** associato ad un arco, esteso ai percorsi
  - $cost(\langle n_i, n_j \rangle) \in [0, +\infty[$, arco $\langle n_i, n_j \rangle$
  - $cost(p)$, percorso $p = \langle n_0, n_1, \ldots, n_k \rangle$:

$$cost(p) = cost(\langle n_0, n_1 \rangle) + \cdots + cost(\langle n_{k-1}, n_k \rangle)$$

  - soluzione **ottimale** p ha costo minimo:
    $\nexists p'$ soluzione con $cost(p') < cost(p)$

Esempio — robot consegne (cont.): ricerca di un percorso da $o103$ a $r123$ nel mondo rappresentato nella figura precedente (solo pos. in ***grassetto***)



- grafo risultante e archi etichettati con il costo

- $N = \{\text{mail}, \text{ts}, o103, b3, o109, \ldots\}$
- $A = \{\langle \text{ts}, \text{mail} \rangle, \langle o103, \text{ts} \rangle, \langle o103, b3 \rangle, \langle o103, o109 \rangle, \ldots\}$
  - $o125$ non ha vicini
  - ts ha come vicino solo mail
  - $o103$ ha i vicini ts, $b3$ e $o109$
  - 3 percorsi da $o103$ a $r123$:
    - $\langle o103, o109, o119, o123, r123 \rangle$
    - $\langle o103, b3, b4, o109, o119, o123, r123 \rangle$
    - $\langle o103, b3, b1, b2, b4, o109, o119, o123, r123 \rangle$
  - Se $o103$ fosse un nodo di partenza e $r123$ obiettivo, ognuno di tali percorsi sarebbe una soluzione
- *Per esercizio*: soluzione ottimale?

Grafi Aciclici Orientati e Alberi

- **ciclo**: percorso non vuoto in cui primo e ultimo nodo coincidono
  - i.e. $\langle n_0, n_1, \ldots, n_k \rangle$ tale che $k > 0$ e $n_0 = n_k$
- **grafo aciclico orientato** (*directed acyclic graph*, DAG): grafo orientato senza cicli
- **albero**: DAG con un solo nodo, la **radice**, senza archi entranti, tutti gli altri ne hanno esattamente uno
  - **foglie**: nodi senza archi uscenti

Grafi costruiti dinamicamente

Gli algoritmi dovranno saper:

1. generare i vicini di un nodo
2. riconoscere i nodi-obiettivo

Misure della complessità dei grafi

fattore di ramificazione

- **uscente** (*forward*) di un nodo: numero di archi uscenti
- **entrante** (*backward*) di un nodo: numero degli archi entranti
  - utili a discutere la *complessità* degli algoritmi
  - si assumeranno *limitati superiormente* da una costante
  - determinano le *dimensioni* del grafo
    - ad es. un albero con fattore uscente b per ogni nodo, ha b^n nodi a distanza di n archi per ciascuno

Esempio — robot consegne (cont.) grafo in figura

- per $o103$ il fattore di ramificazione uscente è 3, mentre il fattore di ramificazione entrante è 0
- per mail 0 e 1, risp.
- per $b3$, 2 e 1

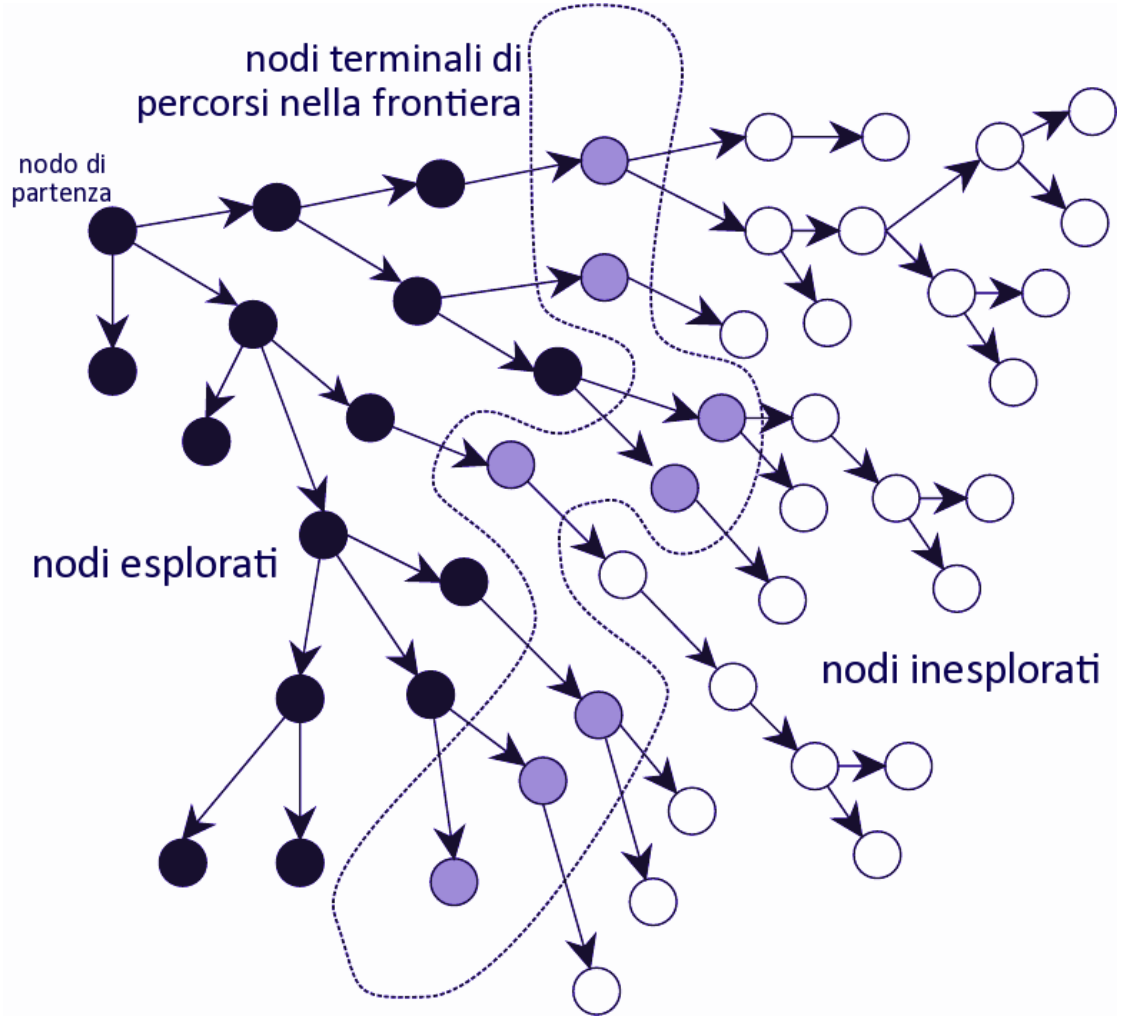
Algoritmo di Ricerca Generico

Algoritmo di Ricerca Generico

Algoritmo indipendente da strategia di ricerca e/o dal grafo

Idea — dato un grafo, si esplorano *incrementalmente* percorsi dai nodi di partenza verso nodi-obiettivo

- struttura dati: **frontiera** (*fringe*) di percorsi già esplorati
  - segmenti iniziali di percorsi completi verso goal
- *inizialmente*: frontiera con percorsi costituiti dai soli *nodi di partenza*
- *successivamente*: *espansione* di percorsi nella frontiera verso nodi inesplorati, fino a incontrare goal
  - si seleziona un percorso (rimuovendolo dalla frontiera)
  - si estende il percorso con ogni arco uscente dall'ultimo nodo
  - si aggiungono alla frontiera i percorsi ottenuti



Ricerca generica usando una frontiera

procedure Search($G, S, goal$)
  Input
    G: grafo con insiemi di nodi N e di archi A
    S: insieme dei nodi di partenza
    $goal$: funzione booleana sugli stati
  Output
    percorso da un elemento di S a un nodo per il quale $goal$ sia vera
    oppure $\perp$ se non ci sono percorsi/soluzioni
  Local
    $Frontier$: insieme di percorsi

  $Frontier \leftarrow \{\langle s \rangle : s \in S\}$
  while $Frontier \neq \emptyset$ **do**
    seleziona e rimuovi $\langle s_0, \ldots, s_k \rangle$ da $Frontier$
    if $goal(s_k)$ **then**
      return $\langle s_0, \ldots, s_k \rangle$
    $Frontier \leftarrow Frontier \cup \{\langle s_0, \ldots, s_k, s \rangle : \langle s_k, s \rangle \in A\}$
  return $\perp$

Osservazioni:

- selezione *non deterministica* ([1])
  - ha un impatto sull'efficienza
  - una strategia particolare determina il percorso da scegliere
- **return** annidato interpretabile come *temporaneo*
  - continuando → strade alternative
- $\perp$ indica che non vi sono (altre) soluzioni
- $goal(s_k)$ testato *dopo* la selezione dalla frontiera,
  non all'aggiunta del nuovo nodo:
  1. a volte esiste un *arco* verso un goal ma *di costo elevato*
     - non sempre conviene restituire il percorso corrispondente:
       potrebbe esserci un percorso di costo inferiore
     - importante nei problemi di ottimizzazione
  2. il test stesso può essere *costoso*
- un percorso che termini con un nodo non-goal senza vicini va rimosso

Strategie di Ricerca Non Informate

Strategie di Ricerca Non Informate

Il problema determina grafo e obiettivo
la **strategia di ricerca** specifica il percorso da selezionare dalla frontiera

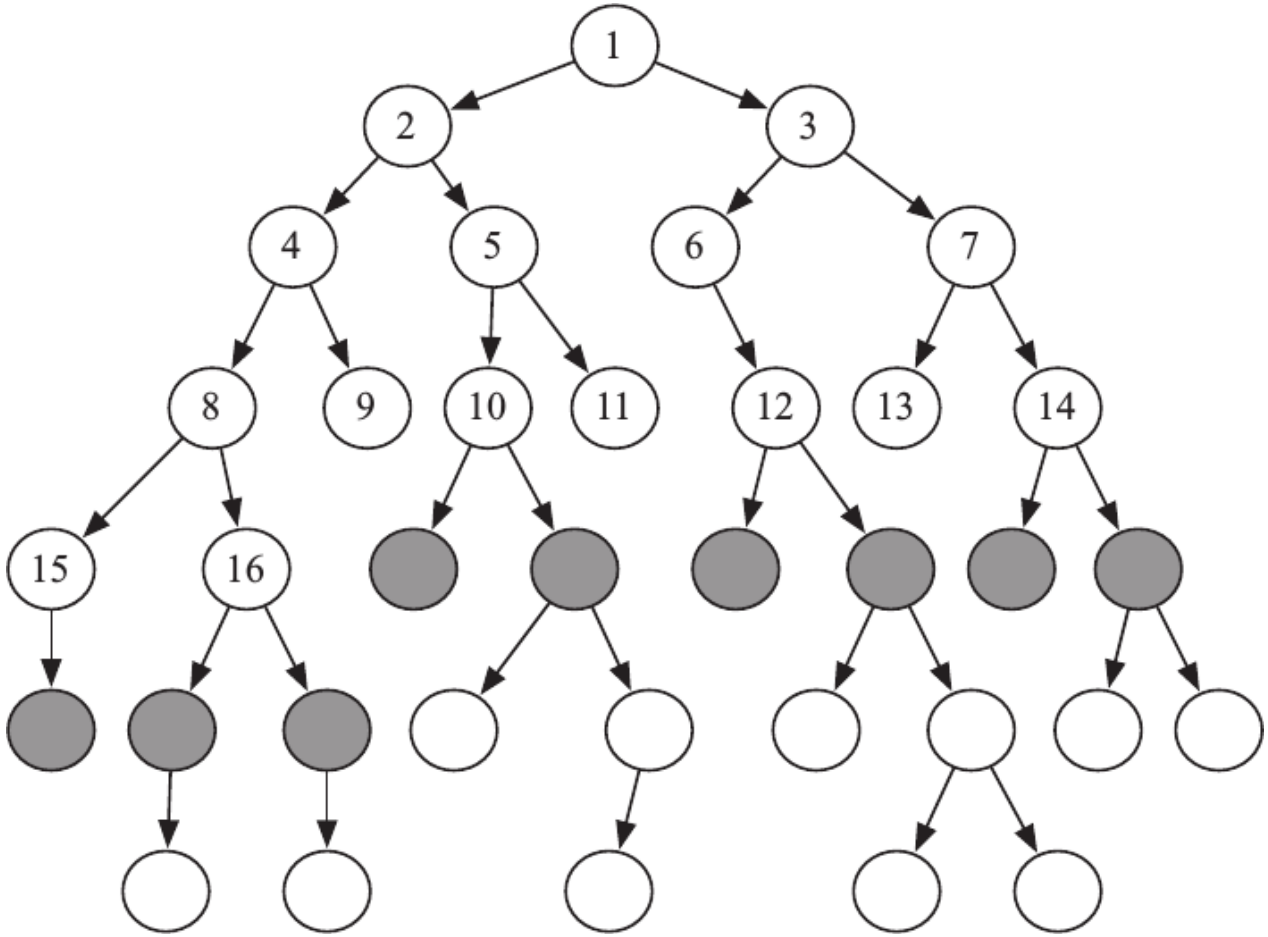
- *strategie* **non informate**:
  non prendono in considerazione la posizione dell'obiettivo
  - costo unitario per ogni arco:
    - *in ampiezza*
    - *in profondità*
    - *iterative deepening* /ˈɪt̬.ə.reɪ.t̬ɪv ˈdiː.pən.ɪŋ/
  - funzione di costo:
    - *costi minimi*

Ricerca in Ampiezza

Nella **ricerca in ampiezza** (BREADTH-FIRST SEARCH, BFS):

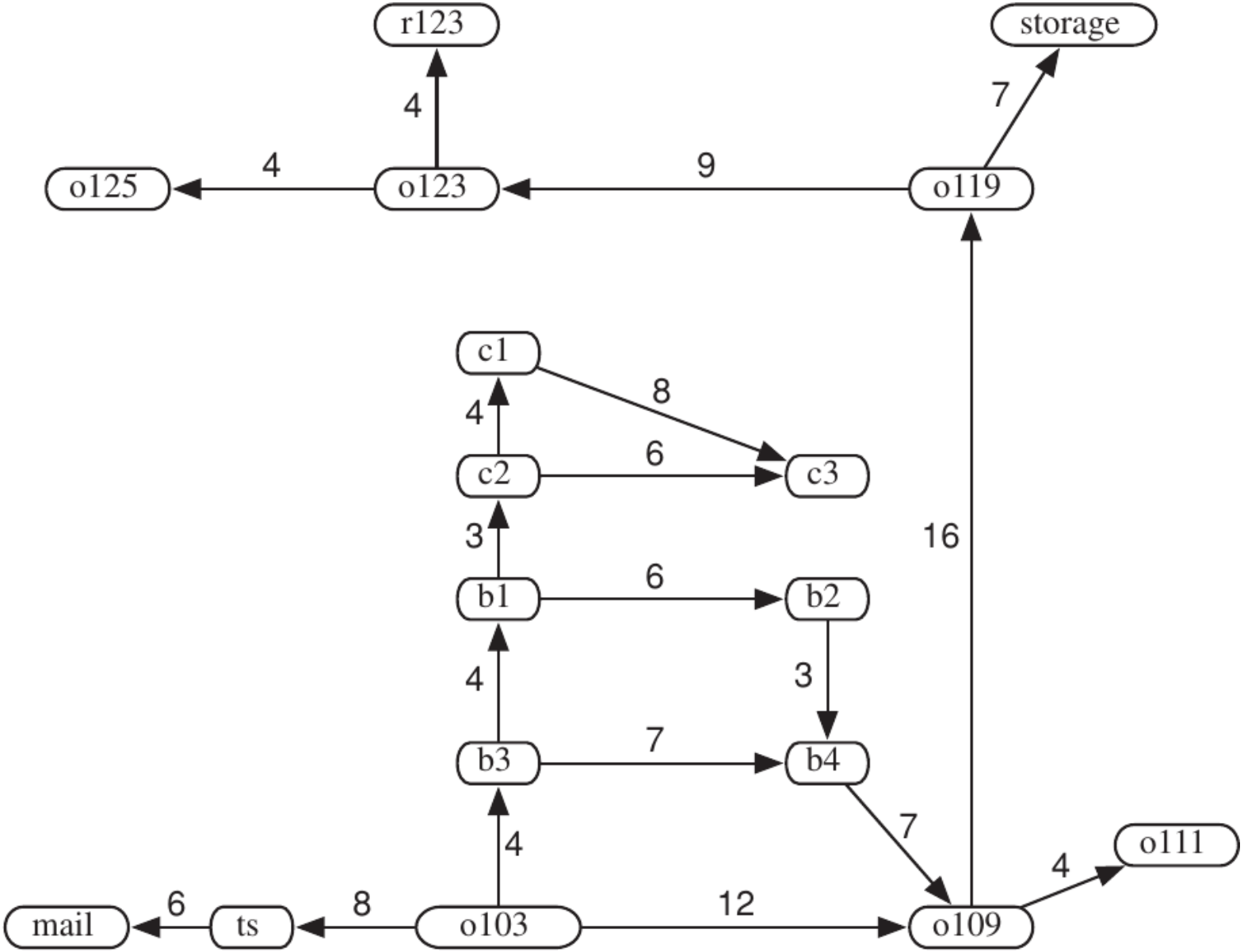
- frontiera implementata con una *coda*,
  struttura FIFO (first-in, first-out)
- si seleziona il *primo* percorso aggiunto
  - percorsi generati nell'ordine del numero di archi contenuti
  - ad ogni passo, si seleziona uno dei percorsi più corti

Esempio — grafo (albero di ricerca): ordine di visita BFS



nodi scuri: terminali dei percorsi della frontiera dopo i primi 16 passi

Esempio — Consideriamo di nuovo il grafo



- $o103$ nodo di *partenza* e $r123$ unico nodo *obiettivo*
  - frontiera: $[\langle o103 \rangle]$
- Estendendo $\langle o103 \rangle$ con i vicini di $o103$:
  - $[\langle o103, \mathrm{ts} \rangle, \langle o103, b3 \rangle, \langle o103, o109 \rangle]$
  - nodi a un arco di distanza da $o013$
- Espandendo questi percorsi di 2 archi, nell'ordine:
  - $[\langle o103, \mathrm{ts}, \mathrm{mail} \rangle, \langle o103, b3, b1 \rangle, \langle o103, b3, b4 \rangle,$ $\langle o103, o109, o111 \rangle, \langle o103, o109, o119 \rangle]$
- Dopo l'espansione dei percorsi di 3 archi precedenti:
  - $[\langle o103, b3, b1, c2 \rangle, \langle o103, b3, b1, b2 \rangle, \langle o103, b3, b4, o109 \rangle,$ $\langle o103, o109, o119, \mathrm{storage} \rangle, \langle o103, o109, o119, o123 \rangle]$
- A ogni passo, percorsi con approssimativamente lo stesso numero di archi (al più uno di differenza)

COMPLESSITÀ

Sia b il fattore di ramificazione:

- se il primo percorso della frontiera ha n archi, ci sono almeno b^{n-1} elementi nella frontiera
  - con n o $n+1$ archi
- quindi complessità in spazio e tempo *esponenziali* nel numero degli archi del percorso di soluzione di lunghezza minima

UTILITÀ

Garanzia di ritrovamento della soluzione quando esiste:

- quella con il minimo numero di archi

BFS *utile* quando:

- non si hanno problemi di spazio
- si cerca una soluzione con numero di archi *minimale*
  - anche se sono poche, ce n'è sempre una di lunghezza minima
- spazio con *percorsi infiniti*: spazio interamente esplorabile

BFS *non utile* quando:

- tutte le soluzioni associate a percorsi lunghi
- è disponibile conoscenza euristica
- il grafo viene generato dinamicamente → per la complessità in spazio

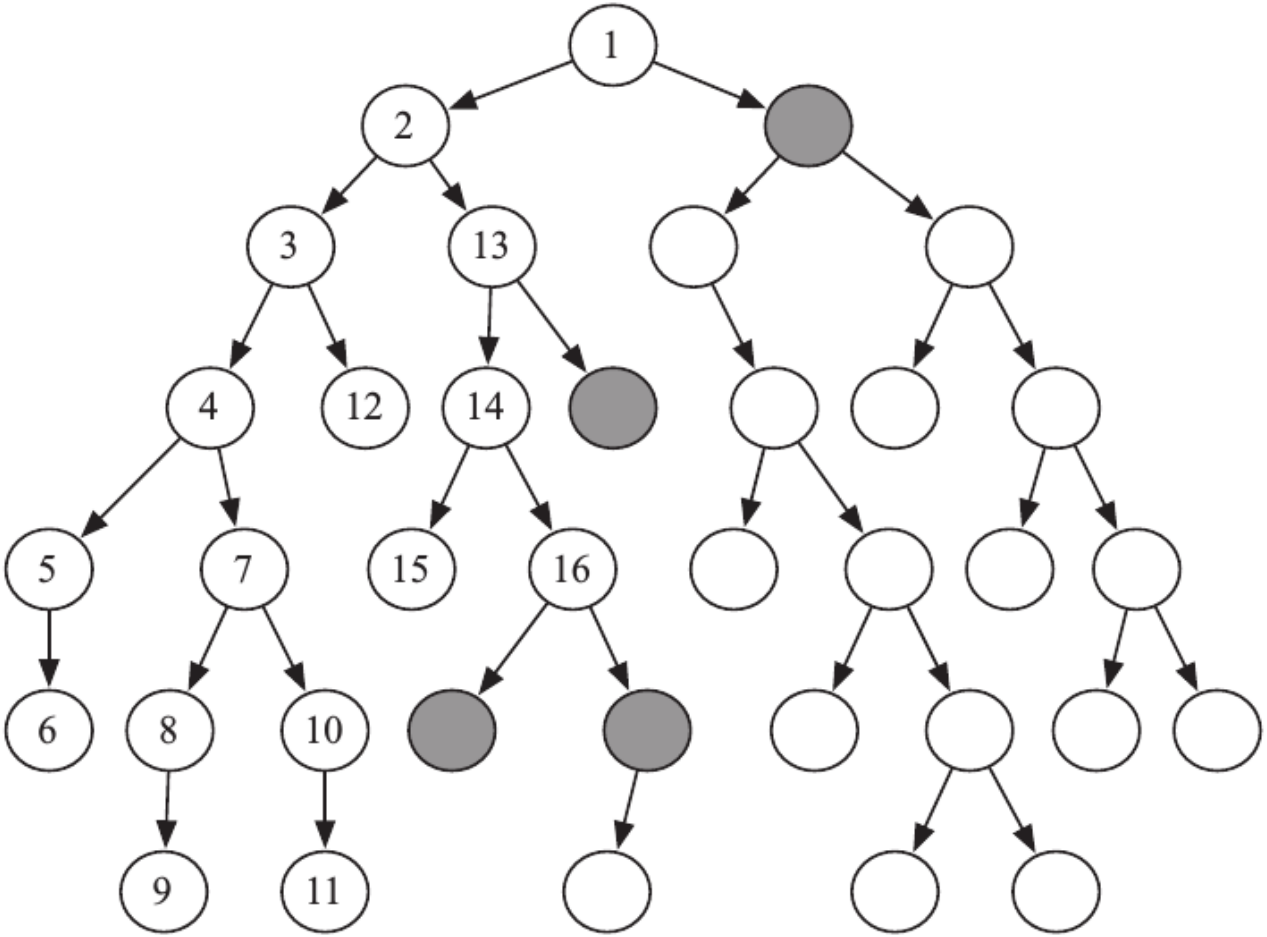
Ricerca in Profondità

Nella **ricerca in profondità** (DEPTH-FIRST SEARCH, DFS):

- frontiera come *pila* (struttura LIFO)
  - elementi aggiunti uno alla volta
  - quello selezionato e prelevato sarà l'ultimo aggiunto

- Partendo dalla radice, nodi considerati ordinati da sinistra a destra
  - vicino più a sinistra aggiunto in cima *per ultimo*
  - ordine di espansione non dipende dalla posizione dei nodi-obiettivo

Esempio — Nel grafo (albero): ordine DFS



nodi scuri: estremità dei percorsi alla frontiera dopo i primi 16 passi:
primi sei nello steso percorso; il nodo 6 non ha vicini → successivo = figlio dell'antenato più in basso con figli da espandere

Backtracking ed efficienza

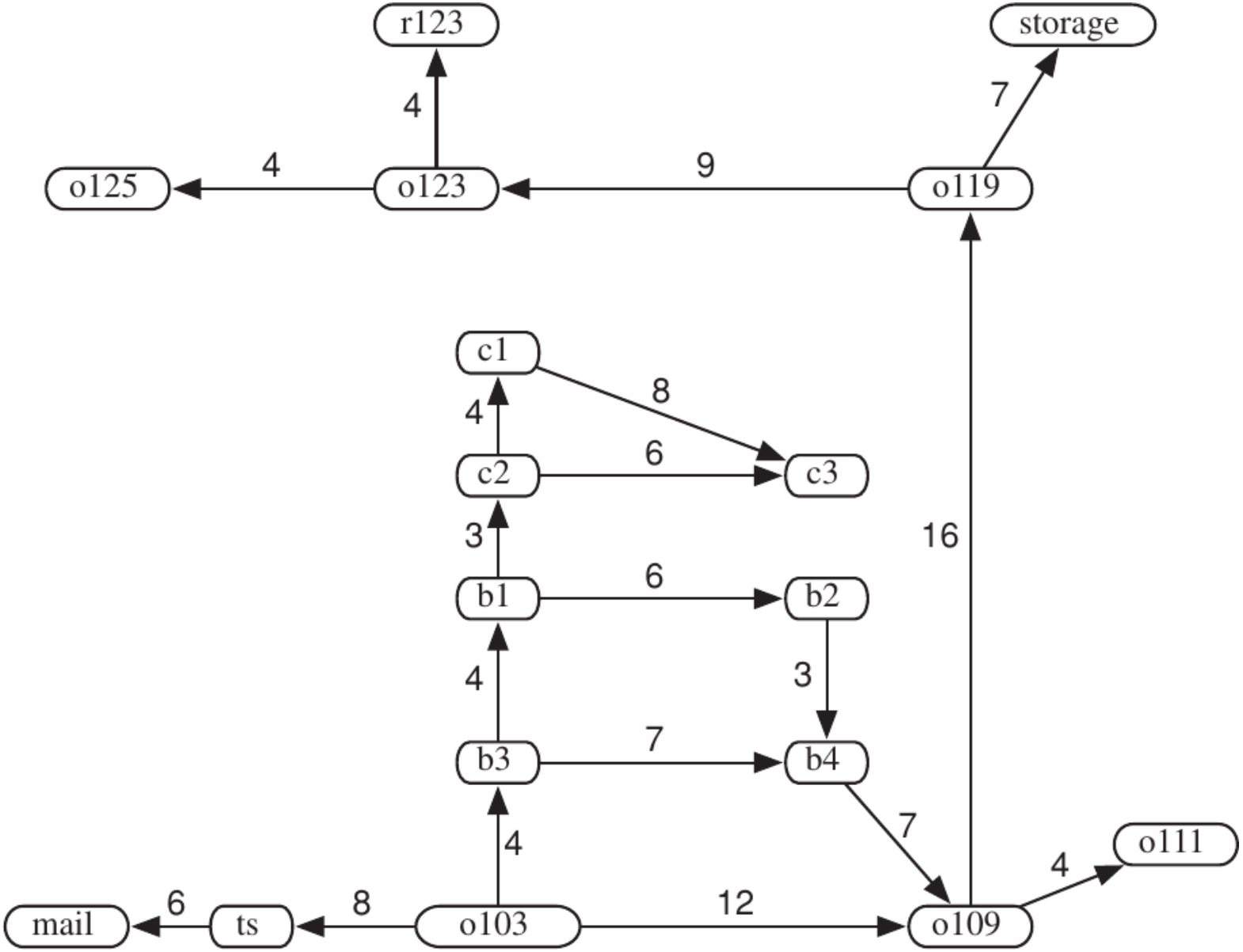
Con uno stack la *ricerca* procede *in profondità*:

- *completamento* di un *singolo percorso* prima di provare alternative
- questo comporta il **backtracking**:
  - si seleziona una prima alternativa per ogni nodo, *tornando indietro* alla successiva solo dopo aver tentato tutti i completamenti
- *ordine di aggiunta* dei vicini alla frontiera/pila non specificato: questo impatta sull'efficienza

grafo con possibili percorsi *infiniti* → l'algoritmo *diverge*:

- se contiene cicli oppure se $|N| = \infty$ (generazione dinamica dei nodi)

Esempio — DFS nel caso precedente: *start* $o103$, *goal* $r123$



- pila (frontiera) come lista di percorsi: *top* = inizio lista

Inizialmente: $[\langle o103 \rangle]$, successivamente:

- Esteso con i 3 vicini:
  - $[\langle o103, \mathrm{ts} \rangle, \langle o103, b3 \rangle, \langle o103, o109 \rangle]$
- Poi si preleva $\langle o103, \mathrm{ts} \rangle$ in cima alla pila, sostituito dalla sua estensione:
  - $[\langle o103, \mathrm{ts}, \mathrm{mail} \rangle, \langle o103, b3 \rangle, \langle o103, o109 \rangle]$
- Quindi, si preleva $\langle o103, \mathrm{ts}, \mathrm{mail} \rangle$
  - insieme di percorsi che lo estendono, vuoto
    - mail non ha vicini
  - quindi, la pila rimane: $[\langle o103, b3 \rangle, \langle o103, o109 \rangle]$
    - seguiti tutti i percorsi da ts (solo 1)
    - avendoli esauriti, si *torna indietro* al successivo elemento della pila
- Si seleziona $\langle o103, b3 \rangle$ rimpiazzato come segue:
  - $[\langle o103, b3, b1 \rangle, \langle o103, b3, b4 \rangle, \langle o103, o109 \rangle]$

- Quindi si seleziona $\langle o103, b3, b1 \rangle$, rimpiazzato con le estensioni:
  - $[\langle o103, b3, b1, c2 \rangle, \langle o103, b3, b1, b2 \rangle, \langle o103, b3, b4 \rangle, \langle o103, o109 \rangle]$
- Espandendo il primo percorso:
  - $[\langle o103, b3, b1, c2, c3 \rangle, \langle o103, b3, b1, c2, c1 \rangle, \langle o103, b3, b1, b2 \rangle, \langle o103, b3, b4 \rangle, \langle o103, o109 \rangle]$
- $c3$ non ha vicini: si torna all'ultima alternativa da seguire, il percorso attraverso $c1$ ...

Proprietà — Se $\langle n_0, \ldots, n_k \rangle$ è il percorso scelto dalla frontiera, ogni altro percorso in essa contenuto ha la forma:

$$\langle n_0, \ldots, n_i, m \rangle$$

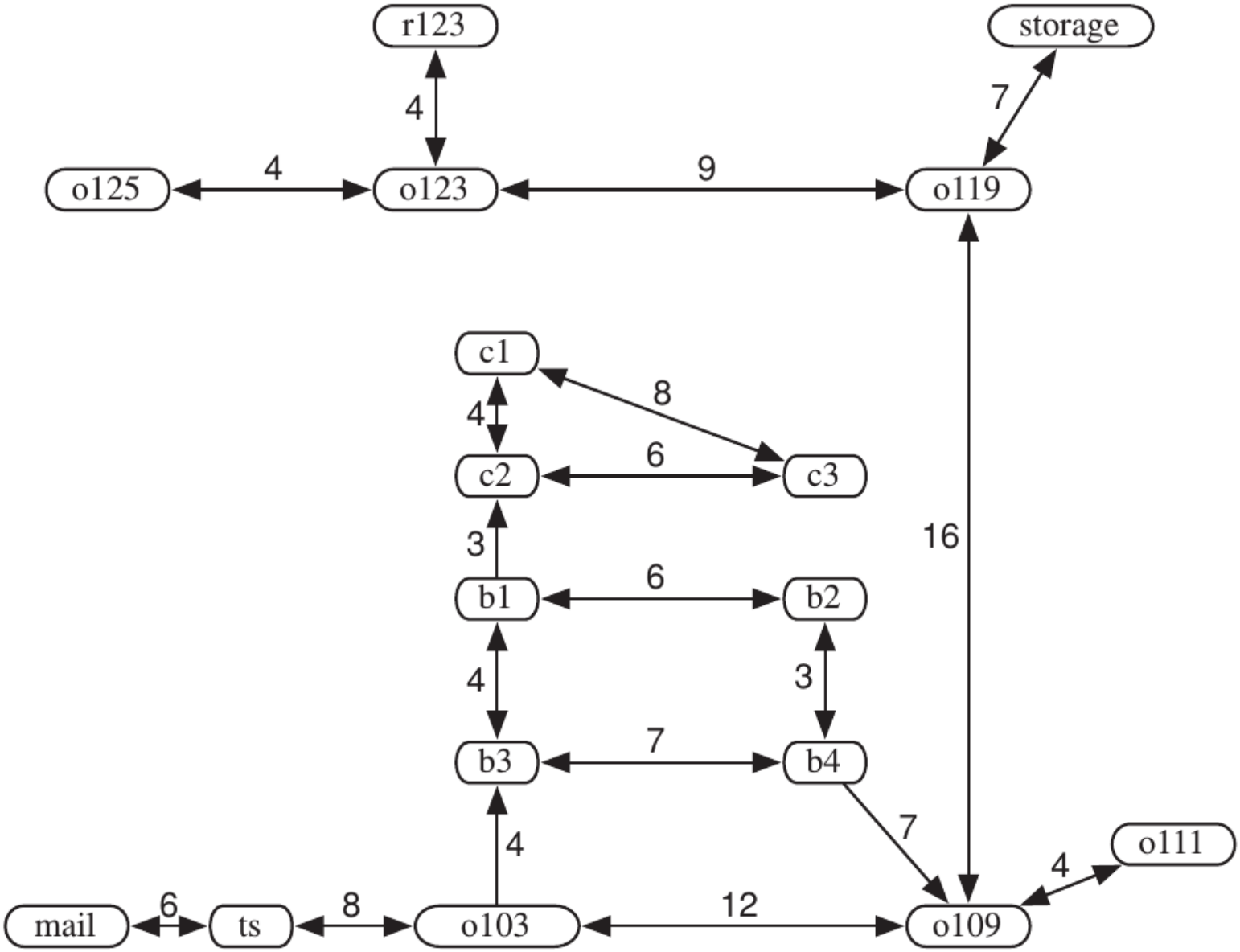
per qualche $i < k$ e nodo m vicino di n_i

- ossia, segue il percorso selezionato per un certo numero di archi e poi ha esattamente un altro nodo

COMPLESSITÀ

- Se b fattore di ramificazione e k lunghezza del primo percorso della lista, ci sono al più *altri* $k(b-1)$ percorsi
  - da ogni nodo, fino a $b-1$ percorsi alternativi
  - quindi spazio ***lineare*** rispetto alla lunghezza del percorso
- Caso ***ottimo***: se c'è una soluzione già sul primo ramo, complessità lineare nella lunghezza del percorso
  - solo elementi nel percorso e loro fratelli
- Caso ***pessimo***: diverge
  - caso di grafi infiniti o con cicli
    - si può rimanere intrappolati in infinite ramificazioni senza trovare una soluzione, anche quando esiste
  - se il grafo è un ***albero finito***, con fattore di ramificazione limitato da b e profondità k, caso pessimo esponenziale: $O(b^k)$

Esempio — nuovo grafo



- percorso infinito iterando tra ts e mail
  - DFS potrebbe andare in loop non considerando alternative via $b3$ o $o109$
- frontiere per le prime 5 iterazioni:
  - $[\langle o103 \rangle]$
  - $[\langle o103, \mathrm{ts} \rangle, \langle o103, b3 \rangle, \langle o103, o109 \rangle]$
  - $[\langle o103, \mathrm{ts}, \mathrm{mail} \rangle, \langle o103, \mathrm{ts}, o103 \rangle, \langle o103, b3 \rangle, \langle o103, o109 \rangle]$
  - $[\langle o103, \mathrm{ts}, \mathrm{mail}, \mathrm{ts} \rangle, \langle o103, \mathrm{ts}, o103 \rangle, \langle o103, b3 \rangle, \langle o103, o109 \rangle]$
  - $[\langle o103, \mathrm{ts}, \mathrm{mail}, \mathrm{ts}, \mathrm{mail} \rangle, \langle o103, \mathrm{ts}, \mathrm{mail}, \mathrm{ts}, o103 \rangle,$ $\langle o103, \mathrm{ts}, o103 \rangle, \langle o103, b3 \rangle, \langle o103, o109 \rangle]$

Si può migliorare la ricerca *evitando* i percorsi con *cicli* ($\rightarrow$ potatura)

Ordine

Ordine di aggiunta dei vicini alla frontiera

- *statico*: prefissato
- *dinamico*: dipende dall'obiettivo

UTILITÀ

- DFS *appropriata* in caso
  - di limitazioni di spazio
  - di presenza di molteplici soluzioni, anche se costituite da percorsi lunghi
    - ideale quando *tutti* portano a una soluzione
    - oppure quando l'ordine di aggiunta dei vicini possa essere regolato in modo da trovare una soluzione al *primo tentativo*
      - senza backtracking
- DFS *inefficiente* quando:
  - sono possibili percorsi infiniti
    - grafo infinito o contenente cicli
  - pur esistendo soluzioni alternative poco profonde (più a destra), la ricerca si attarda su percorsi più lunghi

Iterative Deepening

Obiettivo: combinare l'efficienza in spazio di DFS con l'ottimalità di BFS

- *Idea*: non memorizzare ma ricalcolare gli elementi della frontiera di BFS
  - con DFS si usa meno spazio
- DFS all'interno delle iterazioni di BFS:
  - ricerca fino a una *profondità limitata*, eliminando precedenti computazioni e ripartendo, se necessario
    - si parte da 1 come lunghezza max di percorso,
      poi tutti quelli di profondità 2,
      quindi quelli di profondità 3, ecc...
  - se esiste, una soluzione verrà trovata,
    esplorando percorsi in ordine di lunghezza:
    - quello con meno archi sarà individuato per primo

Fallimento della ricerca:

- *innaturale*: per raggiungimento del limite di profondità
  - la ricerca riparte, avendo incrementato il limite
- *naturale*: spazio di ricerca esaurito (non coinvolge limiti)
  - inutile ritentare: non esiste soluzione a nessun livello di profondità

procedure ID_search(G,S,$goal$)
Input
  G: grafo con insiemi di nodi N e di archi A
  S: insieme di nodi di partenza
  $goal$: funzione Booleana sugli stati
Output
  cammino da S a un nodo per il quale $goal$ sia vero ovvero $\perp$ altrimenti
Locali
  hit_depth_bound: Boolean
  $bound$: integer

```
procedure Depth_bounded_search(⟨n_0, ..., n_k⟩, b)
    Input
        ⟨n_0, ..., n_k⟩: percorso
        b: integer, b ≥ 0
    Output
        percorso fino a nodo-obiettivo di lunghezza k + b
    if b > 0 then
        for each ⟨n_k, n⟩ ∈ A do
            res ← Depth_bounded_search(⟨n_0, ..., n_k, n⟩, b − 1)
            if res percorso then
                return res
    else if goal(n_k) then
        return ⟨n_0, ..., n_k⟩
    else if n_k ha vicini then // fallimento innaturale
        hit_depth_bound ← true
```

```
bound ← 0
repeat
   hit_depth_bound ← false
   res ← Depth_bounded_search({⟨s⟩ : s ∈ S}, bound)
   if res percorso then
      return res
   bound ← bound + 1
until not hit_depth_bound
return ⊥ // fallimento naturale
```

Osservazioni su `Depth_bounded_search`

- Implementa una DFS (ricorsiva con stack) con profondità limitata
  - trova percorsi di lunghezza $k + b$, dove k è la lunghezza del percorso dal nodo di partenza e $b \geq 0$
    - chiamata a profondità crescenti
- Percorsi trovati nello stesso ordine della BFS
  - controlla di aver trovato un obiettivo solo per $b = 0$:
    no soluzioni per limiti inferiori
- Per assicurare di fallire quando anche la BFS fallirebbe, tiene traccia dei casi in cui un limite maggiore potrebbe aiutare a trovare una soluzione
  - Fallisce *naturalmente*, se ha esaurito tutto lo spazio di ricerca e non sono stati tagliati percorsi per limite raggiunto
    - ci si può fermare restituendo $\perp$
  - se hit_depth_bound, falsa alla chiamata, diventa vera alla fine, il limite può essere incrementato per la successiva iterata

Complessità 4

Problema: spreco di calcolo a ogni passo

- non così grave se il fattore di ramificazione b è grande (cfr. seguito)
- *tempo*: dato $b > 1$ costante, in una ricerca con limite k
  - profondità k: b^k nodi, generati una sola volta;
    profondità $k-1$: b^{k-1} nodi generati 2 volte;
    profondità $k-2$: b^{k-2} nodi generati 3 volte;
    ...
    profondità 1: b^1 nodi generati k volte

- totale nodi generati:
  $b^k + 2b^{k-1} + 3b^{k-2} + \cdots + kb$
  $= \quad b^k(1 + 2b^{-1} + 3b^{-2} + \cdots + kb^{1-k})$
  $\leq \quad b^k \sum_{i=1}^{\infty} ib^{(1-i)} = b^k \left(\frac{b}{b-1}\right)^2$
  - la BFS espande $b^k(\frac{b}{b-1}) - \frac{1}{b-1}$ nodi
  - quindi ID ha un overhead asintotico di $b/(b-1)$ volte il costo dell'espansione della BFS
    - per $b = 2$: overhead 2
    - per $b = 3$: overhead 1.5
- algoritmo $O(b^k)$ e non ci può essere una migliore ricerca non informata
- !!! se b è vicino a 1, analisi problematica: denominatore vicino a 0

Ricerca a Costo Minimo

Archi con *costi* associati non unitari:

- → si cerca la soluzione di *minimo costo totale*
  - es. robot delle consegne: in base a distanze
    e/o le risorse necessarie a svolgere l'azione rappresentata da un arco

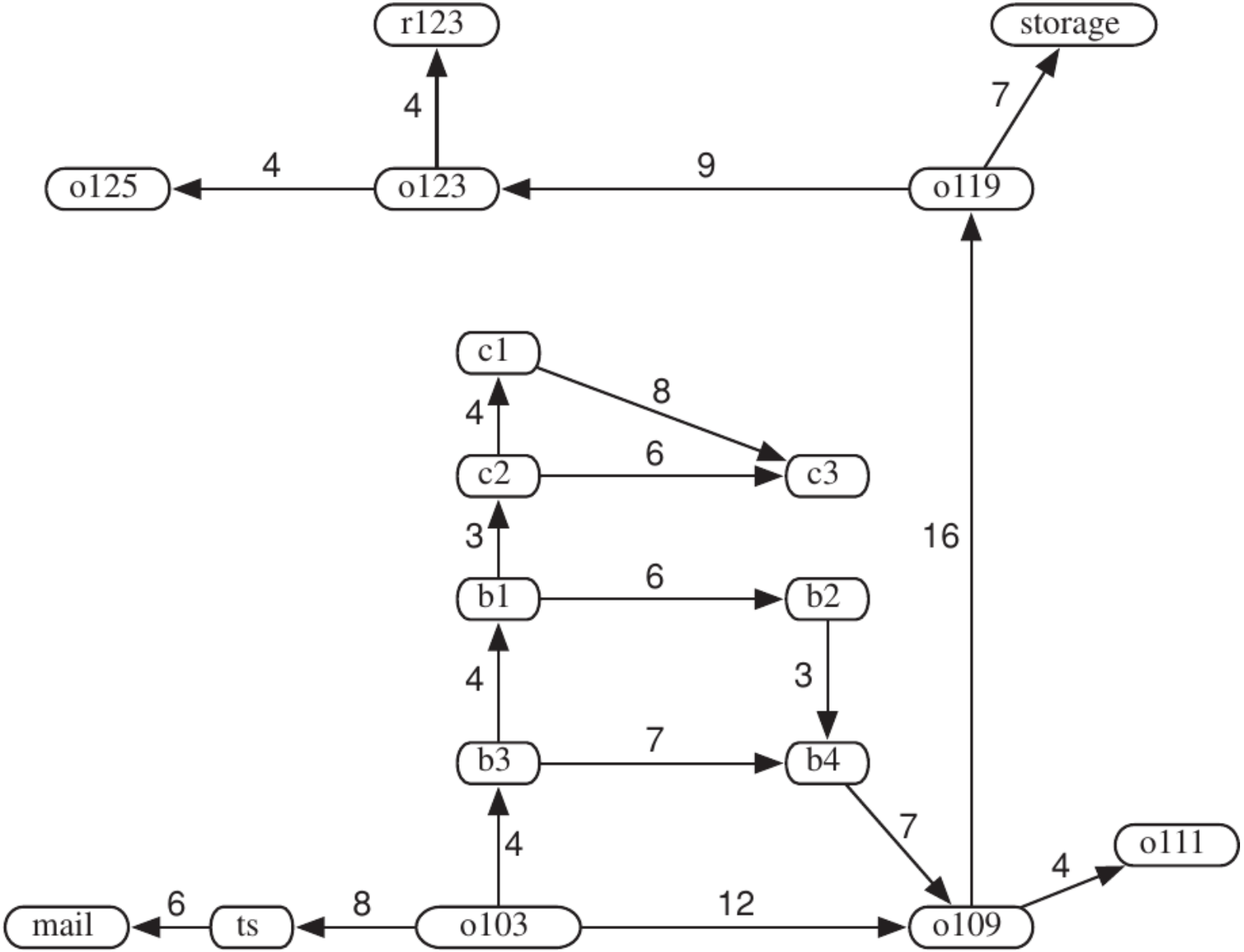
Gli algoritmi precedenti NON garantiscono soluzioni di costo minimo:

- costo non preso in considerazione
  - BFS minimizza solo il *numero* degli archi
    - potrebbe esistere una soluzione alternativa
      con un percorso *più lungo* di *costo inferiore*

LOWEST COST-FIRST SEARCH (LCFS):

- BFS con selezione dei percorsi di costo minimo
  - implementato con *coda con priorità* come frontiera ordinata da $cost$

Esempio — Ancora sul grafo:



Denotando in percorsi attraverso il loro ultimo nodo, con il relativo costo come pedice:

- frontiera iniziale: $[o103_0]$
- prossimo passo: $[b3_4, ts_8, o109_{12}]$
- selezionato quello di $b3$, nuova frontiera: $[b1_8, ts_8, b4_{11}, o109_{12}]$
- seleziona il percorso di $b1$, nuova frontiera: $[ts_8, c2_{11}, b4_{11}, o109_{12}, b2_{14}]$
- selezionato il percorso su ts, si ha: $[c2_{11}, b4_{11}, o109_{12}, \mathrm{mail}_{14}, b2_{14}]$
- quindi tocca a $c2$ e così via ...

Calcolabilità — Ottimalità — Complessità

- Fattore di ramificazione *finito* se i *costi* sono *limitati* inferiormente da una costante positiva: garantita la soluzione ottimale, se esiste
  - primo percorso trovato termina in un nodo-obiettivo
  - ottimale perché si procede in ordine di costo
    - se ne esistesse una migliore sarebbe stata già trovata
- Senza limite inferiore percorsi *infiniti* possibili
  - ad es. nodi $n_0, n_1, n_2, \ldots$ con $\forall i > 0$: $\langle n_{i-1}, n_i \rangle$ di costo $1/2^i$
  - esistono infiniti percorsi $\langle n_0, n_1, n_2, \ldots, n_k \rangle$, con costo < 1
  - se $\exists \langle n_0, g \rangle,\ goal(g)$, con costo ≥ 1, non verrebbe mai selezionato
    - cfr. paradosso di Zenone: *Achille e la tartaruga*
- Complessità *esponenziale* in spazio e tempo (come BFS)
  - genera tutti i percorsi con costo inferiore a quello della soluzione

Ricerca Euristica

Ricerca Euristica

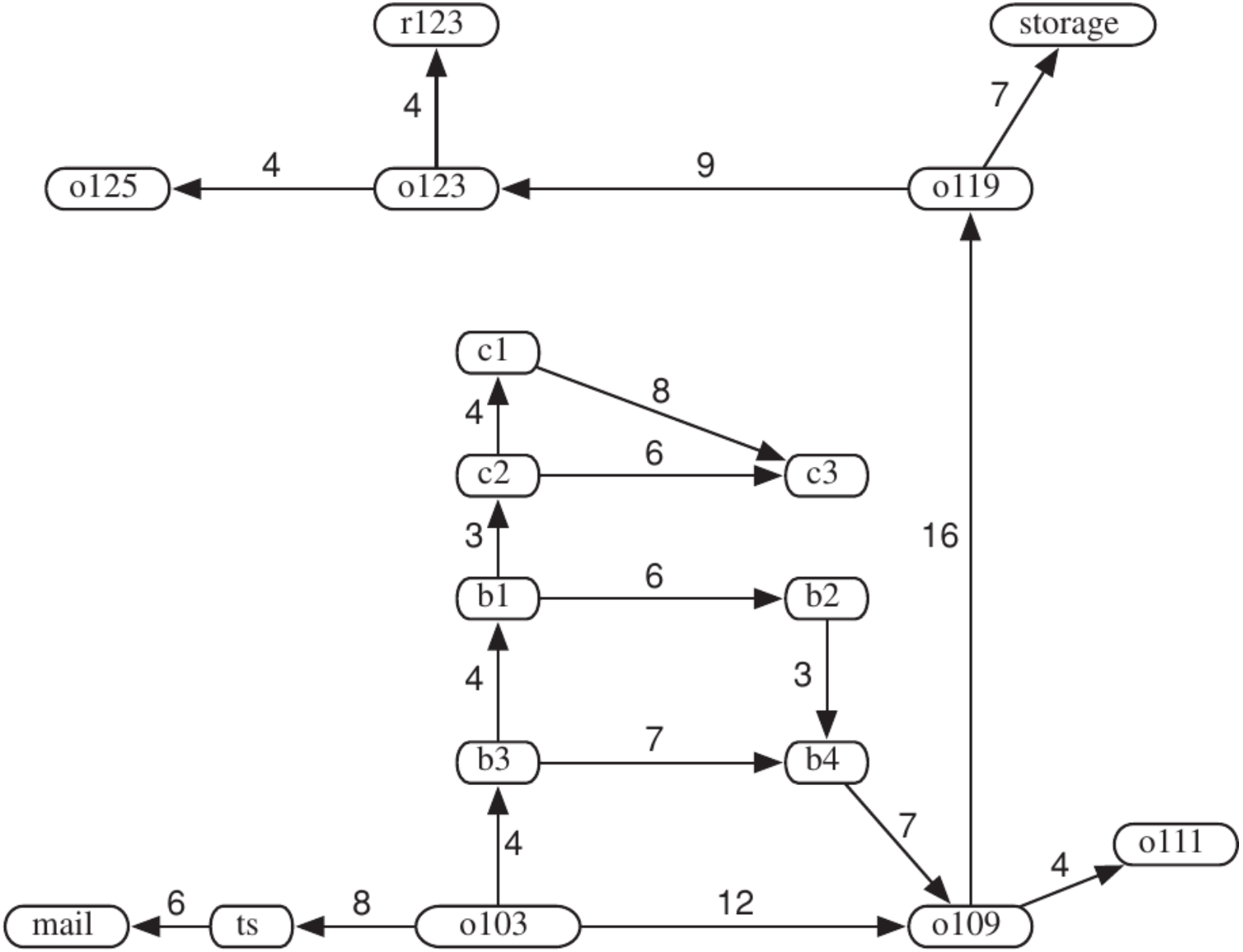
Ricerca che prende in considerazione *informazioni sull'obiettivo* nella selezione dei nodi attraverso una **funzione euristica** h:

- a ogni nodo n associa un numero reale *non-negativo*
  - stima del costo minimale d'un percorso fino a un nodo-goal
- h **ammissibile** se *sottostima* il costo:
  - $h(n)$ minore o uguale rispetto al costo effettivo minimale del percorso

Si sfrutta informazione immediatamente disponibile:

- *compromesso* tra efficienza e accuratezza della stima
- tipicamente: risolvendo il problema in forma semplificata, si usano i costi relativi come euristica per risolvere il problema originale

Esempio — problema precedente



- *euristica*: distanza in linea retta tra nodo e obiettivo vicino
  - si assumano i seguenti valori:

$h(mail)$	$=$	26	$h(o109)$	$=$	24	$h(o123)$	$=$	4
$h(b1)$	$=$	13	$h(b4)$	$=$	18	$h(c3)$	$=$	12
$h(ts)$	$=$	23	$h(o111)$	$=$	27	$h(o125)$	$=$	6
$h(b2)$	$=$	15	$h(c1)$	$=$	6	$h(storage)$	$=$	12
$h(o103)$	$=$	21	$h(o119)$	$=$	11	$h(r123)$	$=$	0
$h(b3)$	$=$	17	$h(c2)$	$=$	10			

- h ammissibile:
  - *esatta* per $o123$
  - *sottostimata* per $b1$: sembra vicino al goal (13),
    ma raggiungere l'obiettivo costa in realtà 45
  - *ingannevole* per $c1$: sembra vicino (6) ma non permette di raggiungere l'obiettivo

Esempio — robot delle consegne
con rappresentazione degli stati che includa i pacchi da consegnare

- *costo* pari alla distanza totale percorsa per consegnare tutti i pacchi
- *euristica possibile*: distanza più grande di un pacco dalla sua destinazione
  - se il robot può trasportare un solo pacco, l'euristica può sommare le distanze dei pacchi dalle loro destinazioni
  - se può portarne di più, questa non sarà una sottostima

h *estesa* al caso dei percorsi (non vuoti):

$$h(\langle n_0, \ldots, n_k \rangle) = h(n_k)$$

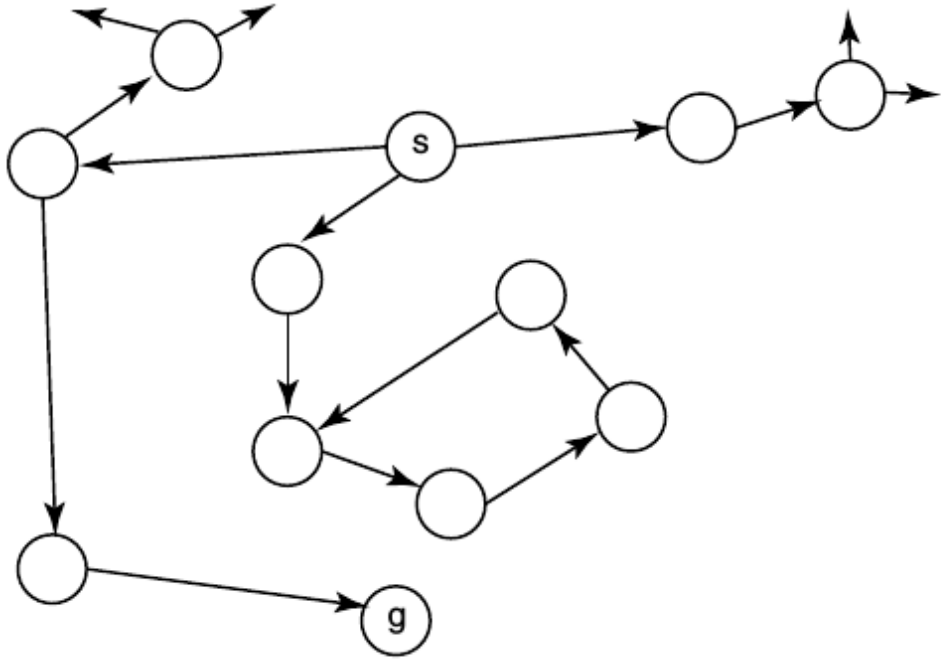
DFS Euristica

- h per ordinare i vicini aggiunti alla pila/frontiera della DFS:
  - vicini aggiunti in modo che il *migliore* vada *in cima*
  - scelta *locale*:
    - esplora i percorsi che estendono quello selezionato prima di tentarne altri
  - stessi problemi della DFS

Greedy best-First Search, GBFS

- scelta del percorso della frontiera con valore di h minimo
  - !! potrebbe seguire cammini promettenti (vicini all'obiettivo secondo h) che potrebbero continuare ad allungarsi indefinitamente

Esempio — Si consideri il grafo:



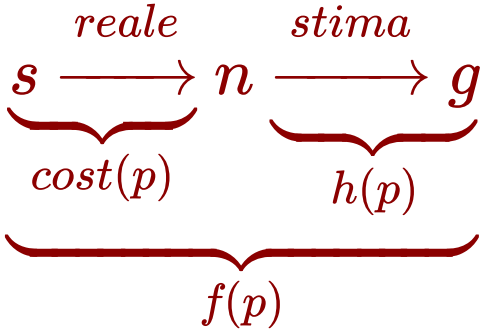
- *scopo*: percorso minimo da s a g
- *costo*: lunghezza dell'arco
- h: distanza Euclidea da g
  - DFS euristica: seleziona sempre il nodo sotto s divergendo
  - GBFS: analogamente, dato che tutti i nodi sotto s sembrano buoni, itererà su di essi, non provando mai strade alternative (da s)

A* combina idee da LCFS e GBFS:

- nella selezione del percorso da espandere si considerano:
  - costo del percorso parziale
  - euristica (stima del costo fino a un obiettivo)
- per ogni percorso $p = \langle s, \ldots, n \rangle$ della frontiera,
  stima del costo di un percorso completo (fino a un goal g) che lo estenda

$$f(p) = cost(p) + h(p)$$

  - $cost(p)$ costo di p (partendo da un nodo iniziale s)
  - $h(p)$ stima del costo del cammino successivo da n a g



Implementazione di A*

Schema di algoritmo di ricerca generico:

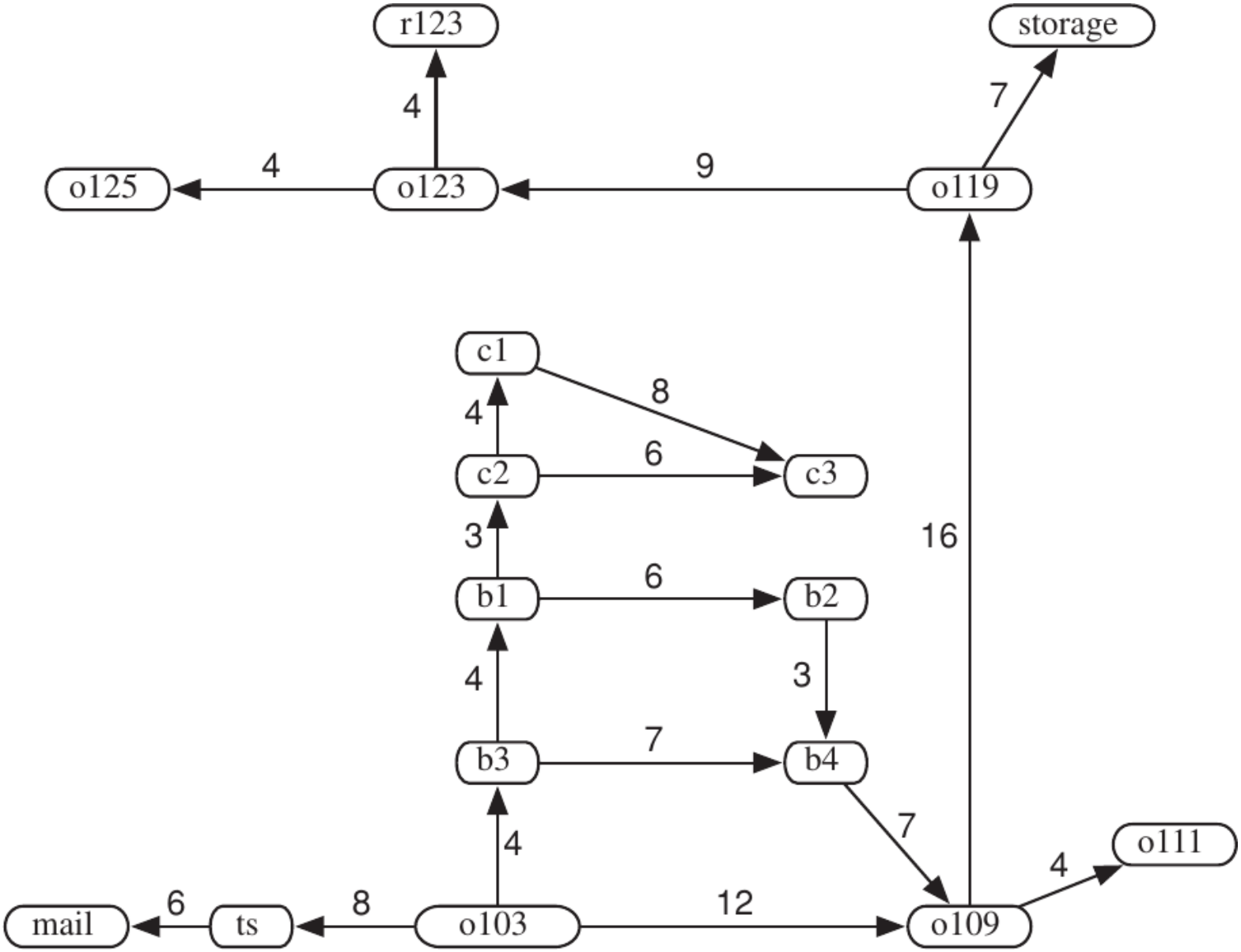
- *frontiera* = coda con priorità, ordinata da $f(p)$
- $h(n)$ *ammissibile*
  → $f(p)$ non sovrastima il costo d'un percorso completo che includa p

A* può seguire molti percorsi ma *alla fine* ne trova uno di costo minimo

- molti possono sembrare (provvisoriamente) di costo inferiore
- migliora le prestazioni degli algoritmi da cui origina (LCFS e GBFS)

Esempio — A* su grafo ed euristica precedenti

$$
\begin{array}{rcrrcrrcr}
h(mail) & = & 26 & h(o109) & = & 24 & h(o123) & = & 4 \\
h(b1) & = & 13 & h(b4) & = & 18 & h(c3) & = & 12 \\
h(ts) & = & 23 & h(o111) & = & 27 & h(o125) & = & 6 \\
h(b2) & = & 15 & h(c1) & = & 6 & h(storage) & = & 12 \\
h(o103) & = & 21 & h(o119) & = & 11 & h(r123) & = & 0 \\
h(b3) & = & 17 & h(c2) & = & 10 & & &
\end{array}
$$


r123
storage
4
7
4
9
o125
o123
o119
c1
8
4
c2
6
c3
16
3
6
b1
b2
4
3
7
b3
b4
7
4
o111
4
6
8
12
mail
ts
o103
o109

(..cont.)

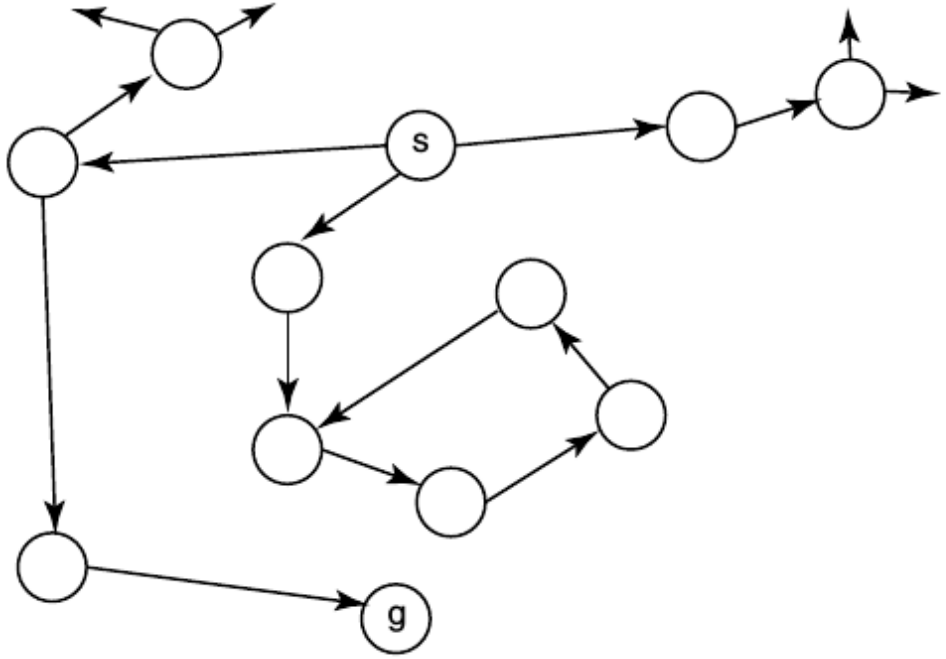
Ogni percorso della ***frontiera*** $p = \langle s, \ldots, n \rangle$ indicato con $n_{f(p)}$

- frontiera iniziale: $[o103_{21}]$ nodo di partenza
  - $h(o103) = 21$ e $cost(\langle\rangle) = 0$ percorso parziale precedente vuoto
- rimpiazzato dai vicini producendo: $[b3_{21}, ts_{31}, o109_{36}]$
  - primo elemento $b3_{21}$ sta per $p = \langle o103, b3 \rangle$ con $f(p) = cost(p) + h(b3) = 4 + 17 = 21$
- si seleziona $b3$, rimpiazzato dai suoi vicini: $[b1_{21}, b4_{29}, ts_{31}, o109_{36}]$
- ora si sostituisce $b1$ con i vicini: $[c2_{21}, b4_{29}, b2_{29}, ts_{31}, o109_{36}]$
- quindi si passa a $c2$ ottenendo: $[c1_{21}, b4_{29}, b2_{29}, c3_{29}, ts_{31}, o109_{36}]$
  - la ricerca sembra perseguire un percorso diretto al nodo obiettivo

(..cont.)

- si seleziona quindi $c1$ ottenendo: $[b4_{29}, b2_{29}, c3_{29}, ts_{31}, c3_{35}, o109_{36}]$
  - !! in coda ***due*** percorsi fino a $c3$
  - quello che non passa per $c1$ ha un valore di f più basso dell'altro
    - in seguito si considererà la possibilità di tagliarne uno
- quale scegliere tra due percorsi con lo stesso valore di f?
  scegliendo quello fino a $b4$, dopo la sostituzione:
  $[b2_{29}, c3_{29}, ts_{31}, c3_{35}, o109_{36}, o109_{42}]$
- si può selezionare $b2$ ma non ha vicini, per cui la frontiera diventa:
  $[c3_{29}, ts_{31}, c3_{35}, o109_{36}, o109_{42}]$
- anche $c3$ viene rimosso senza vicini da aggiungere: $[ts_{31}, c3_{35}, o109_{36}, o109_{42}]$
- ... terminare per **Esercizio**

Esempio — grafo risultante con i metodi precedenti:



- sebbene si indirizzi inizialmente sotto s per via di h, il percorso diventa via via così costoso da far preferire il nodo sul vero percorso ottimale (all'inizio indietro nella coda)

Ammissibilità degli Algoritmi

Algoritmo ammissibile se esistono soluzioni, ne trova sempre una ottimale *(anche se lo spazio degli stati è infinito)*

Proposizione — A^* ammissibile se:

- fattore di ramificazione finito
- costi degli archi maggiori di un certo $\epsilon > 0$
- h ammissibile
  - non supera il costo minimale di percorsi da ciascun nodo a un obiettivo

Osservazioni

- A^* garantisce che la prima soluzione sarà ottimale
  - anche in presenza di *cicli*
- ma non assicura che ciascun nodo *intermedio* selezionato dalla frontiera sia su un cammino ottimo

Importanza dell'Euristica in A*

Miglioramento dell'*efficienza* in A* dato da h:

- sia c il costo del percorso di costo minimo
- se h ammissibile, A* espande ogni percorso dal nodo di partenza nell'insieme

$$\{p \mid cost(p) + h(p) < c\}$$

più alcuni percorsi nell'insieme

$$\{p \mid cost(p) + h(p) = c\}$$

- per l'efficienza di A*, h migliore se riduce la cardinalità del primo insieme

IDA*

IDA* applica ITERATIVE DEEPENING ad A*

- effettua ripetute DFS limitate con *limite* sul valore di $f(n)$
  - invece della profondità / numero di archi
- limite iniziale: $f(s)$
  - s nodo di partenza con il minimo valore di h

Osservazioni

- IDA* come DFS limitata
  ma non espande un percorso con costo-f maggiore del limite corrente
  - se la ricerca fallisce in modo *innaturale*:
    nuovo limite ← minimo valore di f che abbia superato il precedente
- IDA* lavora sugli stessi nodi di A* ma li ricalcola con la DFS invece di memorizzarli
- per soluzioni sub-ottimali, si può usare il limite: $\delta + \min f()$

Progettazione della Funzione Euristica

Euristica ammissibile funzione non negativa h su N, con $h(n)$ mai superiore al costo reale del percorso più breve da n a un nodo-goal

Standard per definire un'euristica:

- trovare una soluzione di un problema più semplice, con *meno vincoli*
  - spesso molto *più facile* da risolvere
  - soluzione ottimale ha costo inferiore a una del problema complesso
    - ogni soluzione del primo lo è anche per il secondo
  - ad es. *problemi spaziali*
    - *costo* = distanza
    - problema che *vincolata* a passare per dati archi (ad es. strade)
    - euristica ammissibile: *distanza Euclidea* tra due nodi
      - problema *senza* i vincoli suddetti

Esempio — robot consegna

- *stati* comprendenti le consegne da fare
- *costo* = distanza totale per tutte le consegne
- *euristica* possibile = massimo tra:
  1. distanza massima per consegne non ancora a destinazione ma non trasportate
     - per ciascuna: distanza dalla sua posizione + distanza da questa fino alla destinazione
  2. distanza alla destinazione più lontana per le consegne trasportate

     - tale massimo non rappresenta una sovrastima
       - soluzione per il problema più semplice che non considera muri e tutte le altre consegne tranne la più difficile
       - è appropriato: vanno consegnate quelle trasportate, passate a prendere quelle ancora non caricate e portate a destinazione
     - se il robot può portare una sola consegna, un'euristica dovrebbe sommare le distanze per il trasporto di ognuna più la distanza dalla più vicina
       - non è detto che sarà effettuata per prima (serve a garantire l'ammissibilità)

Esempio — navigatore

Minimizzazione del *tempo*

- *euristica*: distanza in linea retta dalla locazione corrente al goal divisa per la velocità massima
  - assumendo che ci si possa dirigere a destinazione alla massima velocità
- *euristica* (più sofisticata): date velocità massime diverse per autostrade e strade locali, si sceglie il massimo tra
  1. il minimo tempo stimato per andare a destinazione su strade locali lente
  2. il minimo tempo utile a raggiungere un'autostrada da una strada locale, per poi raggiungere un luogo vicino alla destinazione e arrivarci da strada locale

Euristiche che comportano ricerca

- problema in forma semplificata risolubile attraverso una ricerca più semplice rispetto al problema originario
  - ricerca ripetuta anche più volte e per tutti i nodi
- conviene memorizzare (*cache*) tali risultati in un **DB di pattern** che associ ai nodi del problema semplificato il valore dell'euristica
- nel problema semplificato, spesso meno nodi:
  - più nodi originari mappati su uno *stesso* nodo di quello semplificato
    come una *classe d'equivalenza*

Potatura dello Spazio di Ricerca

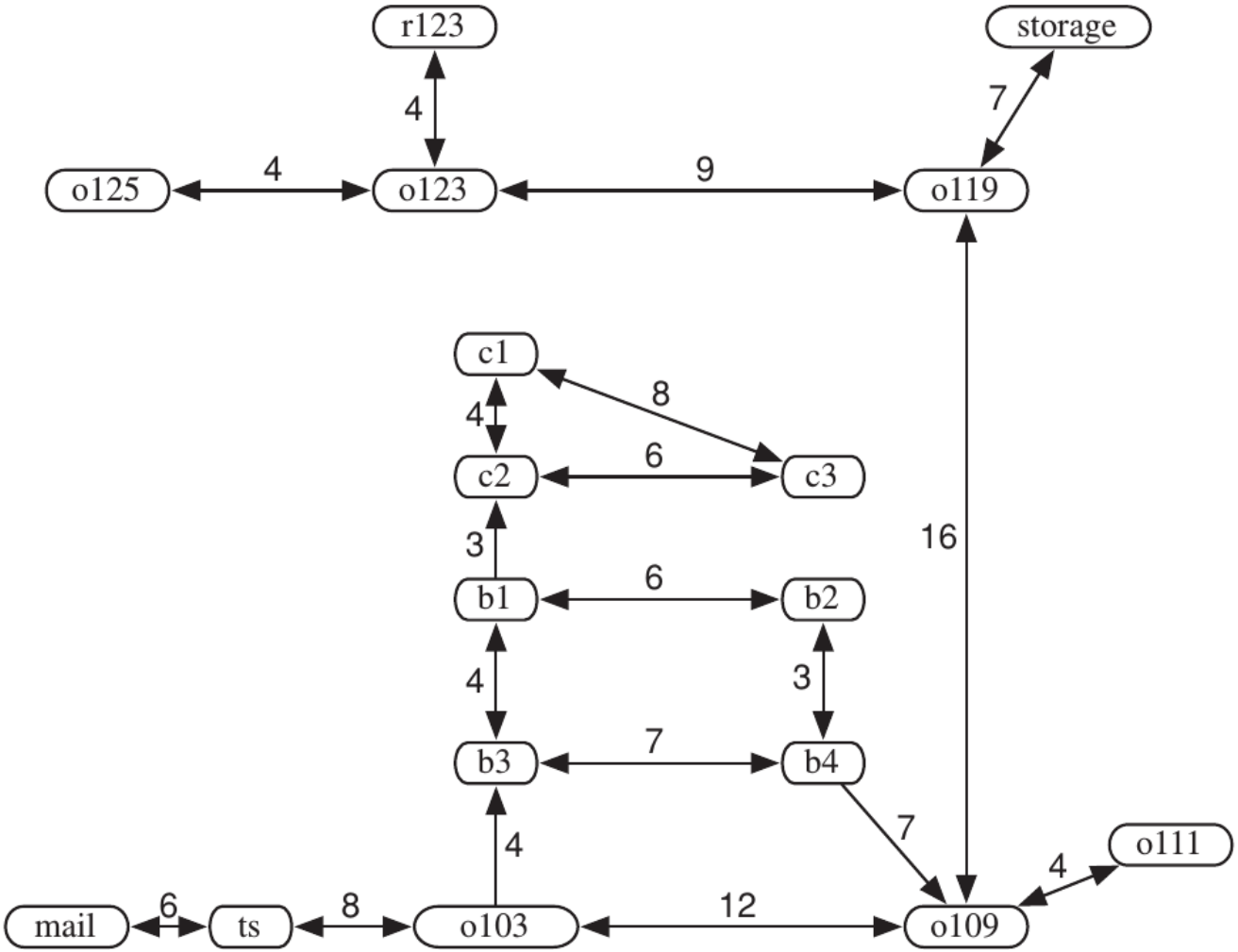
Potatura dello Spazio di Ricerca

Algoritmi migliorabili considerando i diversi percorsi attraverso un nodo

Strategie di **pruning** (*potatura*):

- potare i cicli
  - se si deve trovare un percorso dal costo minimo,
    non si devono considerare i cicli
- potare cammini molteplici
  - per ogni nodo si considera un solo percorso che lo attraversi,
    eliminando tutti gli altri

Alcuni dei metodi visti potrebbero rimanere intrappolati in cicli senza possibilità di uscita anche nel caso di grafi finiti



Garanzia di soluzione con *grafi finiti*:
non prendere in considerazione vicini già contenuti nel percorso

Potatura cicli (*cycle*/*loop pruning*, CP), controllo addizionale preventivo:
dato un nodo da aggiungere, se ne testa l'occorrenza nel percorso

- aggiunti alla frontiera solo percorsi $\langle s_0, \ldots, s_k, s \rangle$,
  con $s \notin \{s_0, \ldots, s_k\}$

In alternativa: controllo *dopo* la selezione del nodo

Overhead della potatura di cicli

La complessità della potatura dipende dal metodo di ricerca usato:

- *costante* per metodi con un solo percorso di lavoro, memorizzato come insieme
  - funzione *hash*
  - *bit* associato a ogni nodo, *acceso* quando viene aggiunto a un percorso e *spento* in caso di backtracking
    - basterà non espandere nodi con il bit acceso
    - funziona perché si lavora su un solo percorso
- *lineare* nella lunghezza del percorso interessato per metodi con più percorsi (esponenziali in spazio)
  - si evita di aggiungere al percorso parziale un nodo già presente

Potatura di Percorsi Multipli

Spesso più di un percorso porta a uno stesso nodo

Idea *eliminare dalla frontiera ogni percorso che porti a un nodo per il quale ne esista già un altro*

Multiple-Path Pruning (MPP)

- implementato gestendo una lista di nodi terminali di percorsi già espansi, detta **closed list** o *explored set*
  - inizialmente vuota
  - selezionando un percorso $\langle n_0, \ldots, n_k \rangle$
    - se n_k è già nella lista esso può essere scartato
    - altrimenti si aggiunge n_k alla lista e si prosegue

non garantisce che il percorso di costo minimo non sia eliminato

Garanzia di soluzioni ottimali

alternative:

- assicurare che il *primo* percorso trovato per un dato nodo sia ottimale
  - gli altri si possono *eliminare*
- se $p = \langle s, \ldots, n, \ldots, m \rangle$ nella frontiera,
  trovando $p' = \langle s, \ldots, n \rangle$ meno costoso della parte fino a n in p, si può:
  - *eliminare* p
  - *sostituire* in p tale parte con p'

Algoritmi di Ricerca e Potatura Multipla

In LcFS, il primo percorso verso un dato nodo (al momento della selezione dalla frontiera) è quello di costo minimo:

- potare altri percorsi non elimina un cammino a costo minimo per il nodo
- quindi tagliare percorsi successivi assicura comunque di poter trovare una soluzione ottimale

A^* non garantisce che quando si seleziona un percorso verso un nodo per la prima volta, questo sia quello di costo minimo:

- ciò è garantito dal teorema di ammissibilità SOLO per i percorsi che portano a nodi-goal
  - per quelli verso altri nodi dipende dalle *proprietà dell'euristica*

Consistenza e Monotonicità

Euristica consistente, h non negativa che soddisfa il vincolo:

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

per ogni coppia di nodi n e n'

- se $h(g) = 0$ per ogni goal g, allora h consistente non sovrastimerà mai il costo dei percorsi da un nodo verso un obiettivo

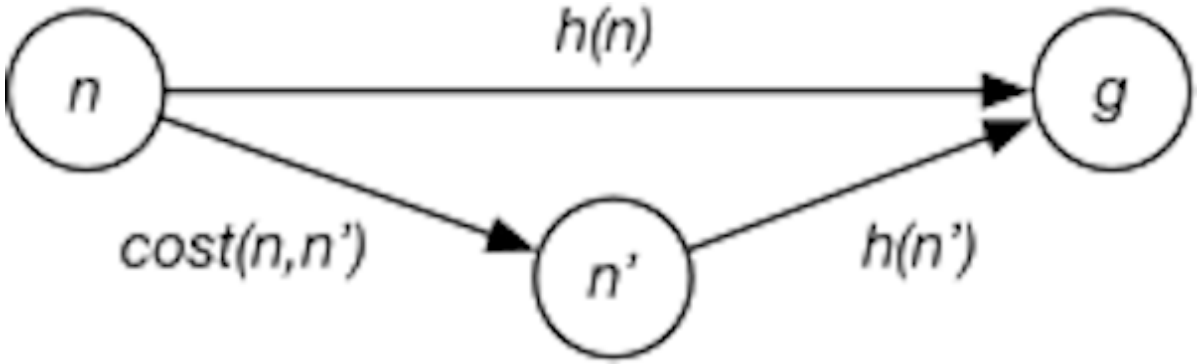
Consistenza garantita se h soddisfa la **restrizione di monotonicità**:

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

per ogni arco $\langle n, n' \rangle$

- più facile da testare: dipende solo dagli archi, non dalle coppie di nodi

Proprietà in termini di **disuguaglianza triangolare**:



- *consistenza*: costo stimato del cammino da n al goal g non maggiore della stima di quello che obbliga al passaggio da n'
  - ad es. distanza Euclidea consistente
  - anche le soluzioni di problemi semplificati tipicamente inducono euristiche consistenti
- *restrizione monotona*: i valori di f dei cammini dalla frontiera sono monotonicamente non decrescenti
  - espandendo la frontiera tali valori possono solo crescere

Proposizione — Data un'euristica consistente, MPP non impedisce ad A^* di trovare una soluzione ottimale

- cioè nelle condizioni della proposizione sull'ammissibilità di A^*, che garantiscono il ritrovamento della soluzione ottima da parte di A^*, se l'euristica è consistente, anche A^* con MPP la potrà trovare

Nella pratica, A^* include MPP per default:

- se si usa A^* senza potatura, ciò andrebbe esplicitato
- è compito del progettista definire un'euristica consistente in modo da permettere il ritrovamento di percorsi ottimali

MPP vs CP

- MPP *più generale* della potatura dei cicli:
  - ciclo come altro percorso verso un dato nodo, destinato a essere potato
- MPP preferibile con metodi *in ampiezza*
  - virtualmente tutti i nodi considerati vanno memorizzati
- Potatura dei cicli preferibile con le strategie *in profondità*:
  - memorizzazione → rischio di crescita esponenziale
- MPP non appropriata per IDA*, meglio CP
  - spazio per la lista chiusa maggiore di quello richiesto per A*
    - venendo meno lo scopo dell'ID
    - in domini con più percorsi verso un nodo, IDA* risulta meno efficace

COMPLESSITÀ

MPP realizzata:

- in tempo *costante* su grafi *espliciti*
  - con un bit per nodo per il quale sia stato trovato un percorso
  - con funzione hash
- in tempo *logaritmico* (nel numero di nodi espansi, opportunamente indicizzati) se il grafo viene generato *dinamicamente*
  - salvando la lista chiusa di tutti i nodi espansi

Consuntivo sulle Strategie di Ricerca

Tabella di sintesi:

Strategia di Ricerca	Selezione da Frontiera	Soluzione Garantita	Complessità in Spazio
DFS	ultimo nodo aggiunto	No	lineare
BFS	primo nodo aggiunto	con meno archi	esponenziale
ID	–	con meno archi	lineare
GBFS	$h(p)$ minimale	No	esponenziale
LCFS	$cost(p)$ minimale	costo minimo	esponenziale
A*	$cost(p) + h(p)$ minimale	costo minimo	esponenziale
IDA*	–	costo minimo	lineare

Completezza degli Algoritmi

Algoritmo di ricerca **completo** garantisce la soluzione quando esiste

- le strategie che trovano percorsi con #archi/costo minimi sono complete
  - *caso pessimo*: tempo esponenziale nel #archi dei percorsi esplorati
  - algoritmi completi con minore complessità?
    dipende dalla risposta alla domanda

$$P \neq NP\,?$$

  - algoritmi che non garantiscono la terminazione → complessità infinita

Strategie di Ricerca più Sofisticate

Strategie di Ricerca più Sofisticate

Raffinamenti possibili delle strategie presentate:

- metodo DF *branch-and-bound*
  - euristico e garantisce soluzioni ottimali
  - con i risparmi in termini di spazio delle strategie in profondità
- metodi basati sulla *riduzione* dei problemi in un numero di problemi di minore complessità
- uso della *programmazione dinamica* nella ricerca euristica

BRANCH-AND-BOUND combina l'efficienza in spazio delle strategie in profondità con informazione euristica

- efficace quando esistono molti cammini verso i/l goal
- si assume $h(n)$ ammissibile

Idea *memorizzare percorso di costo minimo trovato verso un goal e il suo costo che diventa un limite, $bound$:*

- se si trova un percorso p tale che $cost(p) + h(p) \geq bound$, allora p può essere eliminato
- un percorso completo migliore del precedente cammino ottimale va a rimpiazzarlo facendo aggiornare anche $bound$
- si continua poi a cercare un'eventuale soluzione migliore

Si genera una sequenza di soluzioni via via migliori fino a quella finale

- tipicamente combinato con la ricerca in profondità, che risparmia spazio
- implementato come DFS limitata,
  con $bound$ definito in termini di *costo* del percorso
  - tende a ridursi al ritrovamento di percorsi più corti
  - memorizza il percorso di costo minimo trovato e lo restituisce alla fine

procedure DFBranchAndBound($G, s, goal, h, bound_0$)

Input

G: grafo con nodi N e archi A

s: nodo di partenza

$goal$: funzione di test dei nodi

h: euristica sui nodi

$bound_0$: limite iniziale (∞ se non indicato)

Output

percorso di costo minimo da s a un nodo obiettivo se esiste una soluzione di costo minore di $bound_0$, oppure $\perp$

Locali

$best\_path$: percorso o $\perp$

$bound$: numero reale non negativo

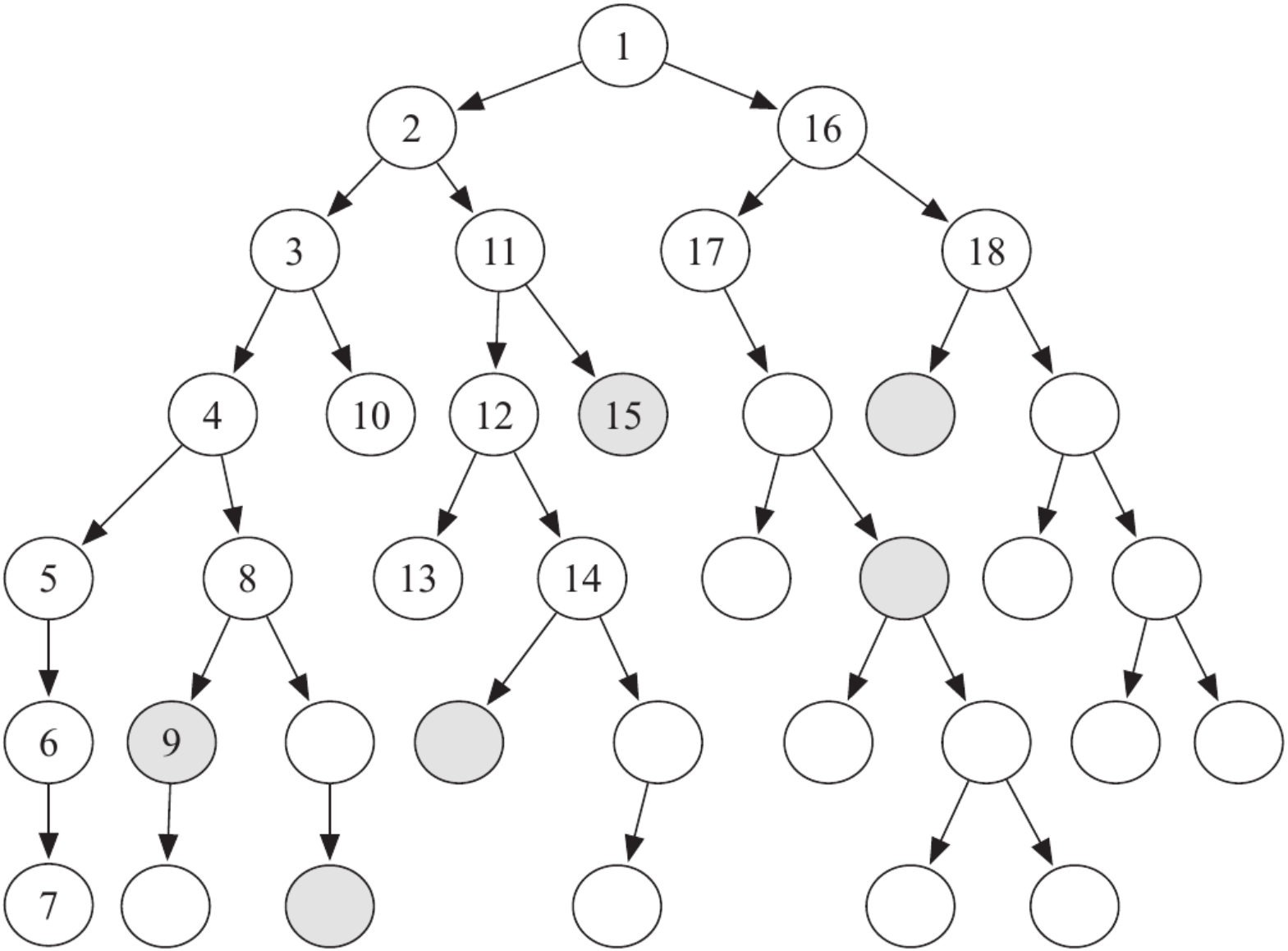
```
procedure cbsearch(⟨n_0, ..., n_k⟩)
    if cost(⟨n_0, ... n_k⟩) + h(n_k) < bound then
        if goal(n_k) then
            best_path ← ⟨n_0, ..., n_k⟩
            bound ← cost(⟨n_0, ..., n_k⟩)
        else
            for each ⟨n_k, n⟩ ∈ A do
                cbsearch(⟨n_0, ..., n_k, n⟩)

best_path ← ⊥
bound ← bound_0
cbsearch(⟨s⟩)
return best_path
```

Osservazioni

- `cbsearch`, ***cost-bounded search***, comunica attraverso variabili globali
  - inizialmente, $bound$ impostata a $bound_0$, stima per eccesso del costo d'una soluzione ottimale
- `DFBranchAndBound` restituirà, se esiste, una soluzione ottimale con costo inferiore a $bound_0$
  - se $bound_0$ supera di poco il costo minimo, non espanderà più archi di A*
  - $bound_0$ tale da far eliminare percorsi di costo maggiore
    - trovato un percorso completo, esplora solo percorsi con valore di f inferiore a quello del percorso trovato
    - esattamente i percorsi esplorati da A* quando trova una soluzione
  - se restituisce $\bot$
    - con $bound_0 = \infty$: non ci sono soluzioni
    - con $bound_0$ finito: non ci sono soluzioni di costo inferiore a $bound_0$
- Combinato con ***ID*** può incrementare il limite fino a trovare una soluzione ovvero può mostrare che non c'è soluzione

Esempio — Considerato: (nodi scuri = obiettivi)



(..continua)

- Si supponga che:
  - ogni arco ha costo 1 e non ci sia informazione euristica
    - $\forall n : h(n) = 0$
  - $bound_0 = \infty$
  - si sceglie sempre il nodo più a sinistra
- Si noti l'ordine di test dei nodi (via $goal()$)
  - nodi non numerati non testati
- Il sotto-albero del nodo 5 non ha goal e viene esplorato
  - o fino a profondità $bound_0$ se valore finito
- Il nodo 9 è un goal
  - costo = $5 \longrightarrow$ nuovo limite
    - prenderà in considerazione solo percorsi con costo inferiore a 5

(..continua)

- Anche il 15 è un goal
  - costo = $3 \longrightarrow$ nuovo limite
- Non ci sono altri nodi obiettivo → restituisce il percorso fino al nodo 15
  - ottimale
  - l'altro di pari costo viene tagliato:
    - non si controllano i figli del nodo 18
- Si potrebbe usare l'euristica disponibile per tagliare parti dello spazio come in A^*

Ricerca *simmetrica* ammessa se:

1. il numero dei goal è *finito*
2. per ogni nodo si possono generare i *vicini* nel **grafo inverso** $\{n' : \langle n', n \rangle \in A\}$

ricerca in **avanti** (*forward s.*):

- si comincia da un nodo di partenza e si raggiungono i goal

ricerca all'**indietro** (*backward s.*):

- si comincia da un goal usando il grafo inverso
  - all'inizio i vicini del goal sono tutti gli altri goal $\{n : goal(n)\}$ inclusi nella frontiera
  - si termina trovando il nodo di partenza

Scelta della Direzione

- dimensioni dello spazio di ricerca: b^k
  - b fattore di ramificazione e k dalla lunghezza del cammino
- riducendo questi parametri aumenta l'*efficienza*
  - possibile differenza tra i fattori di ramificazione uscente ed entrante:
    - *principio generale* per la scelta della direzione della ricerca: b inferiore

Ricerca Bidirezionale

Idea: ridurre il tempo di ricerca procedendo in entrambe le direzioni

- quando le due frontiere *si intersecano*,
  ricostruire un singolo cammino dal nodo di partenza al goal

Problema — assicurare che le due frontiere s'incontrino

- ricerca *in profondità*
  - difficile, date le ridotte dimensioni delle frontiere
- ricerca *in ampiezza*
  - garantito

Combinandole in direzioni opposte, si garantirebbe l'intersezione

Fattore per la scelta:

- ricerca in ampiezza: costo del mantenimento *della* frontiera
- ricerca in profondità: costo della ricerca *nella* frontiera per trovare un'intersezione

Risparmi:

- ricerca in ampiezza: tempo proporzionale a b^k
- ricerca simmetrica bidirezionale: nell'ordine di $2b^{k/2}$
  - risparmi esponenziali in tempo,
    anche se la complessità rimane esponenziale
    - si trascura il costo aggiuntivo della ricerca dell'intersezione

Ricerca basata su Isole

Per una maggiore efficienza si sfrutta *conoscenza aggiuntiva*:

- identificare un numero limitato di posizioni, le **isole**,
  dove le ricerche for- e backward si incontrino
  - posizioni nel grafo vincolate ad essere in un percorso risolutivo
    - ad es. percorsi tra stanze situate a piani differenti
      - vincolano la ricerca a passare per l'ascensore su ogni livello,
        fino all'ascensore al livello dell'obiettivo
  - si può decomporre il problema in più *sotto-problemi*
    - ad es., uno per andare dalla stanza iniziale all'ascensore, poi dall'ascensore all'uscita-ascensore presso il piano della destinazione quindi dall'uscita alla stanza destinazione
- Spazi di ricerca ridotti (problemi più piccoli) riducono l'*esplosione combinatoria*

Decomposizione e Identificazione delle Isole

Per trovare un cammino da s e g:

1. identificare un insieme di isole $i_0, \ldots, i_k$
2. trovare percorsi da s a i_0, da i_{j-1} a i_j per ogni $j = 1, \ldots, k$, e da i_k a g
    - ognuno dei sotto-problemi è più semplice del problema generale

- Conoscenza aggiuntiva sulla struttura del grafo
    - isole *inappropriate* possono rendere il problema più difficile (o addirittura impossibile da risolvere)
- Si possono identificare decomposizioni *alternative*

Ricerca in una Gerarchia di Astrazioni

Data la nozione di isola, si possono definire strategie che lavorano a diversi **livelli di astrazione**

- *astrazione* del problema
  - trascurando il maggior numero possibile di dettagli
- si trova una *soluzione parziale*
  - una che richieda meno dettagli per essere trovata
    - ad es., per andare da una stanza ad un'altra potrebbero servire molte azioni del tipo `girare`, ma a un livello di dettaglio più astratto queste possono essere trascurate
  - si risolve il problema per sommi capi, a un livello d'astrazione appropriato, lasciando da risolvere solo problemi minori

Generalizzando la ricerca a isole:

- trovata la soluzione a livello di isola
  si passa a risolvere ricorsivamente i sotto-problemi in modo analogo
  - info sulle soluzioni trovate a livelli più bassi possono servire ai livelli più alti
  - i livelli più alti possono usare tale informazione per riformulare un piano
  - il processo tipicamente non garantisce soluzioni ottimali perché considera solo alcune delle decomposizioni possibili

Con le diverse modalità di *decomposizione*/*astrazione* di problemi

- tutte le strategie di ricerca utilizzabili
  - astrazioni/decomposizioni utili non facili da individuare

Programmazione Dinamica (PD) — metodo generale di ottimizzazione basato sulla conservazione di *soluzioni parziali*:

- una soluzione già prodotta va ritrovata e non ricalcolata

PD come costruzione di un'*euristica perfetta*: $cost\_to\_goal()$

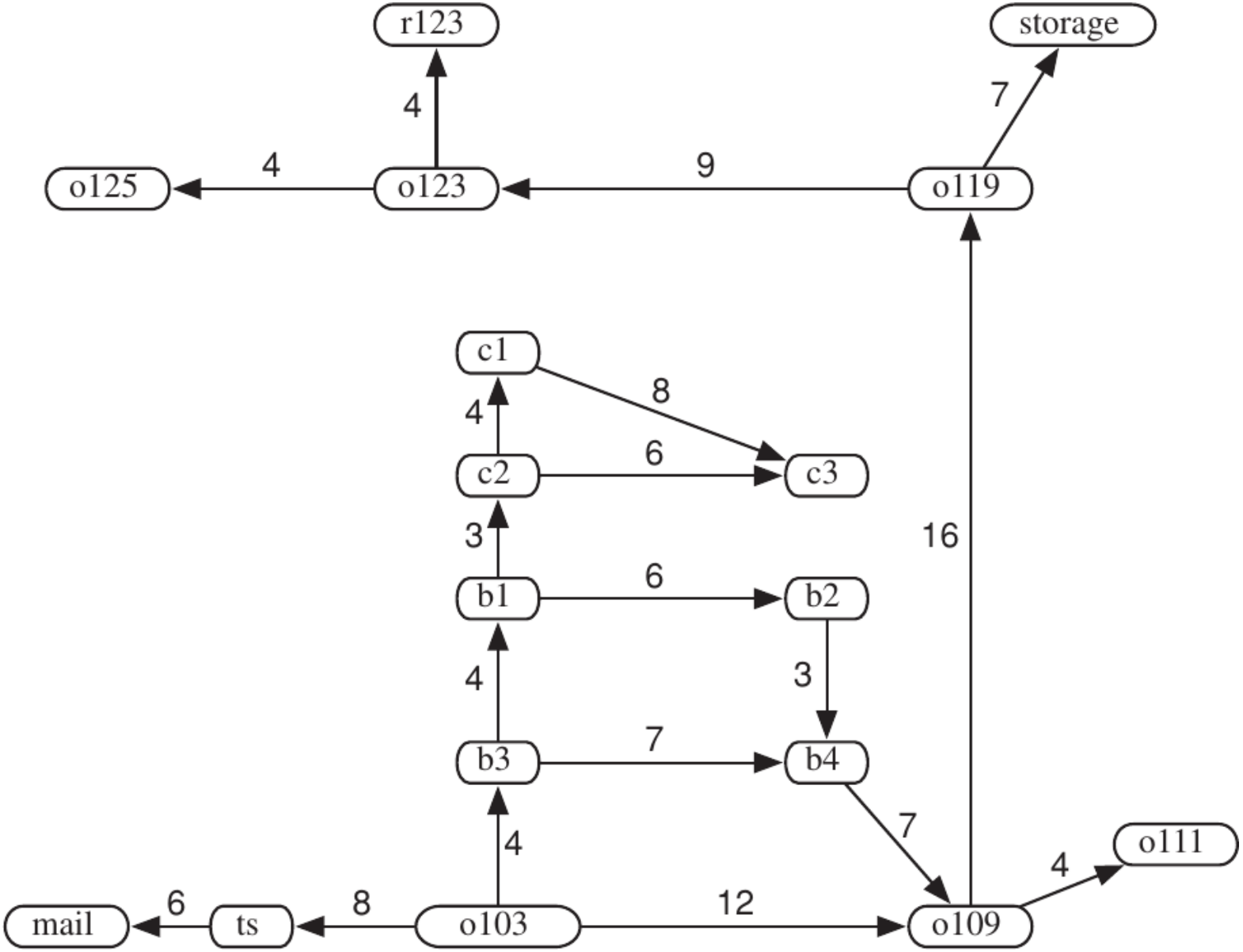
- costo esatto di un cammino completo di costo minimo su *grafi finiti*

$$cost\_to\_goal(n) = \begin{cases} 0 & goal(n) \\ \min\limits_{\langle n,m\rangle \in A} [cost(\langle n,m\rangle) + cost\_to\_goal(m)] & \text{altrimenti} \end{cases}$$

Idea: costruire *offline* una tabella $cost\_to\_goal(n)$ per ogni nodo n:

- mediante LCFS + MPP sul *grafo inverso* a partire dai nodi-obiettivo
  - con tutti gli archi da considerarsi invertiti
- si conserva il valore di $cost\_to\_goal$ per ogni nodo
  - grafo inverso per calcolare i costi della tabella per ogni nodo
- si parte dai goal, cercando all'indietro cammini di costo minimo verso i goal da ogni nodo nel grafo

Esempio — Riprendendo il grafo



- ovviamente, per l'obiettivo $r123$: $cost\_to\_goal(r123) = 0$

Continuando con una ricerca di costo minimo fino a $r123$:

$$\begin{aligned} cost\_to\_goal(o123) &= 4 & cost\_to\_goal(o119) &= 13 \\ cost\_to\_goal(o109) &= 29 & cost\_to\_goal(b4) &= 36 \\ cost\_to\_goal(b2) &= 39 & cost\_to\_goal(o103) &= 41 \\ cost\_to\_goal(b3) &= 43 & cost\_to\_goal(b1) &= 45 \end{aligned}$$

- a questo punto la ricerca si ferma
  1. nodo senza valore in tabella → non c'è un percorso verso un goal
  2. da qualsiasi nodo, si determina rapidamente il prossimo arco su un cammino minimo
     - ad es., per trovare il cammino minimo da $o103$ a $r123$, si confronta $4 + 43$ (costo via $b3$) con $12 + 29$ (costo via $o109$) preferendo quindi $o109$

Policy — specifica dell'arco da considerare per ogni nodo

- **ottimale** se suoi costi mai superiori a costi calcolati con altre policy
- $cost\_to\_goal$ calcolata ***offline*** e usata nella costruzione di una policy:
  - da n si dovrebbe andare al vicino m che minimizza $cost(\langle n, m \rangle) + cost\_to\_goal(m)$
  - questa policy porta sempre a un obiettivo con un camminino di costo minimo partendo da qualsiasi nodo
- alternative:
  - memorizzare il vicino per tutti i nodi ***offline***, sfruttando le associazioni tra nodi nelle decisioni online sulle azioni
  - fornire $cost\_to\_goal$ pre-calcolata e calcolare i vicini online

COMPLESSITÀ

- PD *lineare* in tempo e spazio rispetto alle dimensioni del grafo nella costruzione di $cost\_to\_goal$
  - tipicamente *esponenziali* nella lunghezza del percorso
- Data $cost\_to\_goal$, determinare l'arco migliore richiede tempo *costante* rispetto alla grandezza del grafo
  - numero limitato di vicini per nodo

UTILITÀ

- PD può servire a costruire euristiche per A^* e BRANCH-AND-BOUND:
  - semplificando il problema fino a ottenere spazi di ricerca ridotti
  - trovando in tali spazi soluzioni di lunghezza ottimale
    - → *DB di pattern* usato nell'euristica per il problema originale
- PD utile quando
  - nodi obiettivo *espliciti* (in luogo di $goal()$)
  - soluzione: cammino di *costo minimo*
  - grafo ridotto *finito*, con memoria sufficiente per la tabella
  - l'obiettivo non cambia (frequentemente)
  - policy riusabile più volte per ogni goal
    - ammortizza il costo per produrre la tabella su più istanze del problema
- Criticità:
  - policy da calcolare per ogni diverso goal

Riferimenti

Bibliografia

[1] D. Poole, A. Mackworth: *Artificial Intelligence: Foundations of Computational Agents*. 2nd ed. Cambridge University Press [Ch.3]
[2] D. Poole, A. Mackworth, R. Goebel: *Computational Intelligence: A Logical Approach*. Oxford University Press
[3] S. J. Russell, P. Norvig: *Artificial Intelligence* Pearson. 4th Ed. - cfr. anche ed. Italiana [Cap. 3-4]

Link

[graph_search] http://aispace.org/search/
[State_space_search] https://en.wikipedia.org/wiki/State_space_search
[A*] https://en.wikipedia.org/wiki/A*_search_algorithm
[IDA*] https://en.wikipedia.org/wiki/Iterative_deepening_A*

Note

[≺] consigliata la lettura

[1] cfr. specchietto Non-deterministic Choice in [1]

[versione] 12/10/2022, 10:52:09

Figure tratte da [1] salvo diversa indicazione