Relazione Progetto di Intelligenza Artificiale

Quadrelli S. 870682 Ragusa F. 871243 Nava L. 872353

Indice

1 Specifiche del Progetto

L'agente da noi creato è un pianificatore gerarchico a due livelli che opera in conoscenza incompleta del mondo. Il suo scopo è raccogliere i rifiuti contenuti nei cassonetti sparsi per un mondo in cui l'agente opera. Non è nota la quantità di rifiuti contenuta nei cassonetti.

Il mondo è diviso in varie aree in base alla densità di popolazione. Le aree sono definite in base alla distanza da un punto: l'appartenenza di un cassonetto ad un'area è definita dalla distanza che intercorre tra esso e il punto più vicino tra quello che rappresenti la densità. I rifiuti devono essere trasportati fino ad un centro di raccolta e il camion che li trasporta ha un quantitativo finito di benzina e di carico trasportabile. Pertanto il suddetto camion deve fare rifornimento e andare a depositare i rifiuti ogni volta che il carico trasportato non consente con certezza di svuotare completamente un altro cassonetto.

Dei cassonetti è nota la posizione, l'area di appartenenza e la quantità massima di spazzatura che vi è possibile trovare ma non la quantità di rifiuti effettivamente contenuta.

1.1 Conoscenze iniziali

- Densità di popolazione delle varie zone del mondo.
- Topologia del mondo.
- Posizione dei cassonetti, area a cui appartengono e limite di rifiuti che potrebbe contenere.
- Capacità massima di rifiuti trasportabili.
- Carico attuale di rifiuti.
- Capienza del serbatoio.
- Quantità di benzina attualmente disponibile.
- Posizione centro raccolta.
- Posizione stazione rifornimento.

1.2 Pianificatore gerarchico

- Azione di livello 1: `va`.
- Azioni di livello 0: `va_da_a` , `raccoglie`, `deposita`, `rifornimento`, `scarica`.

1.3 Conoscenza incompleta

- Non si conosce la quantità di spazzatura effettivamente contenuta nei cassonetti.

1.4 Assunzioni

- I cassonetti nelle aree più popolate possono contenere più spazzatura rispetto a quelli con una densità di popolazione minore.

2 Manuale

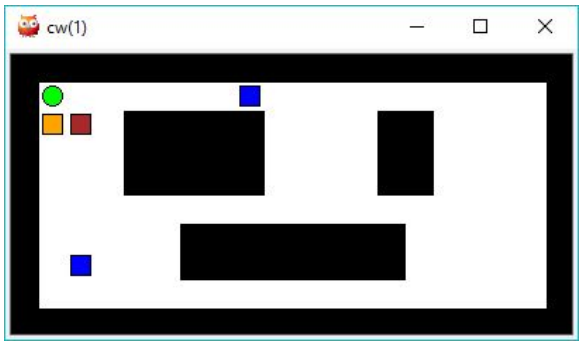
Per eseguire il programma bisogna chiamare la funzione raccolta/9 in spazzino_gerarchico.pl. La funzione richiede i seguenti parametri:

1. Il mondo in cui l'agente deve operare.
2. Il punto di partenza dell'agente.
3. La lista di punti rappresentanti la posizione dei cassonetti.
4. Lista contenente i punti che rappresentano le aree.
5. Lista contenente i limiti massimi di spazzatura delle aree.
6. Il punto che rappresenta la posizione della stazione di rifornimento.
7. Il punto che rappresenta la posizione del centro di raccolta.
8. Un intero che rappresenta la capienza massima del serbatoio.
9. Un intero che rappresenta la capienza massima del camion.

Ne forniamo un esempio. La chiamata:

```
raccolta(cw(1), point(2,2),[point(2,9), point(8,3) ], [point(2,7),
point(10,8)],[4,7], point(3,3),point(3,2),35,15).
```

genera il seguente mondo:



Legenda:

- Cerchio verde: agente
- Quadrato arancione: deposito dei rifiuti
- Quadrato marrone: stazione di rifornimento
- Quadrato blu: cassonetto

Esistono tre casi in cui non è possibile trovare una soluzione:

1. Quando la quantità di rifiuti trasportabile è minore della quantità massima di rifiuti presenti nel cassonetto. Ad esempio:
   `raccolta(cw(1), point(2,2), [point(2,9), point(8,3)], [point(2,7), point(10,8)],[4, 7], point(3,3), point(3,2), 35, 3).`
2. Quando la quantità di benzina non è sufficiente per raggiungere tutti i cassonetti e tornare a fare rifornimento. Ad esempio:
   `raccolta(cw(1), point(2,2), [point(2,9), point(8,3)], [point(2,7), point(10,8)],[4, 7], point(3,3), point(3,2), 5, 15).[1]`
3. Quando la lunghezza della lista delle aree è diversa da quella della lista degli upperbound. Ad esempio:
   `raccolta(cw(1), point(2,2), [point(2,9), point(8,3)], [point(2,7), point(10,8)], [4, 7, 9], point(3,3), point(3,2), 35, 15).`

3 Codice

Commentiamo in questa sezione solo le porzioni di codice da noi modificate e di particolare interesse.

3.1 Fluenti

La lista dei fluenti è stata aggiornata nel seguente modo:

```
type([in(point), deve_prendere(point), deposito(point),
    benzinaio(point), serbatoio(integer), carico(integer),
    fine]:fluent).
%   in(P): l'agente si trova in P
%   deve_prendere(X): l'agente deve prendere un oggetto in X
%   deposito(Q): deve depositare gli oggetti in Q
%   benzinaio(B): deve fare rifornimento in B
%   serbatoio(S): quantita' di benzina S nel serbatoio
%   carico(C): quantita' C di rifiuti gia' raccolti
%   fine: ha finito
```

3.2 Azioni di livello 0

3.2.1 SCARICA

Quest'azione, come si può evincere dal nome, permette all'agente, in fase di pianificazione, di scaricare il carico di rifiuti. Si presuppone che l'agente si trovi nel deposito. Viene aggiunto allo Stato `St` il fluente `carico(0)` e tolto il fluente `carico(C)`. Si usa solo nel caso in cui vi siano altri cassonetti ancora da svuotare altrimenti è utilizzato `deposita`.

```
% azione compiuta quando vi sono ancora cassonetti da svuotare
% ma non si ha spazio a sufficienza per portare a termine la
    raccolta
two_level_planner:add_del(0, scarica, St, [carico(0)],
[carico(C)], 0.5) :-
   member(deve_prendere(_), St),
   member(deposito(P), St),
   member(in(P), St),
   member(carico(C), St).
```

L'azione `va_da_a` ha diverse implementazioni in base alla destinazione, forniamo di seguito la descrizione.

3.2.2 VA_DA_A_CASSONETTO

Quest'azione è usata in fase di pianificazione per raggiungere un cassonetto. Perché ciò sia possibile vengono effettuati due controlli. Inizialmente si controlla che il carico del camion sia minore o uguale alla differenza tra la sua capacità massima e l'upperbound dell'area cui il cassonetto appartiene: ciò è fatto per garantire che una volta giunti ad esso sia possibile svuotarlo completamente.

In seguito si calcola il piano per portare a termine l'azione. Solamente dopo aver ottenuto il costo del piano si effettua il secondo controllo che consiste nel capire se vi è benzina a sufficienza per raggiungere il cassonetto desiderato: si controlla semplicemente che la benzina residua sia maggiore del costo.

I fluenti resi veri aggiornano la posizione dell'agente e il quantitativo di benzina residua.

```
% vai al cassonetto
two_level_planner:add_del(0,va_da_a(P1,P2), St, [in(P2),
    serbatoio(Y)], [in(P1), serbatoio(X)], Cost) :-
% azione che permette di raggiungere un cassonetto del mondo
    decrementando il qunatitativo di benzina disponibile
% controllo che sia possibile svuotare interamente il cassonetto,
    controllando che la capienza del camion - l'upperbound
    dell'area del cassonetto sia <= al carico trasportabile residuo
```

```
   member(carico(Ca), St),
   capienza_camion(Cap),
   cassonetto(P2,Dens,_),
   info_area(Dens, Ub),
   K is Cap - Ub,
   Ca @=< K,
% generazione del piano
   member(deve_prendere(P2),St),
   member(in(P1), St),
   P1 \= P2,
   get_action_plan([in(P1), tager(P2)], va_da_a(P1,P2), _Plan,
       Cost),
% controllo che sia effettivamente possibile raggiungere il
    cassonetto con il livello di benzina attuale
   member(serbatoio(X), St),
   C is round(Cost),
   C @< X,
   Y is X - C.
```

3.2.3 VA_DA_A_BENZINAIO

Quest'azione è usata in fase di pianificazione per raggiungere il benzinaio qualora sia necessario fare rifornimento.

Analogamente all'azione precedente viene calcolato il piano per raggiungere il benzinaio la cui posizione nel mondo è nota e si controlla che vi sia benzina a sufficienza per portare a compimento l'azione.

```
% vai al benzinaio
% azione che permette di raggiungere la stazione di rifornimento
    quando non e' possibile raggiungere alcun cassonetto
two_level_planner:add_del(0,va_da_a(P1,P2), St, [in(P2),
serbatoio(Y)], [in(P1), serbatoio(X)], Cost) :-
% generazione del piano
   member(in(P1), St),
   member(benzinaio(P2), St),
   P1 \= P2,
   get_action_plan([in(P1), tager(P2)], va_da_a(P1,P2), _Plan,
       Cost),
% controllo che sia effettivamente possibile raggiungere il
    benzinaio con il livello di benzina attuale
   member(serbatoio(X), St),
   C is round(Cost),
   Y is X - C,
   Y @>= 0.
```

3.2.4 VA_DA_A_DEPOSITO

Quest'azione è usata in fase di pianificazione per raggiungere il deposito. Anche in questo caso viene calcolato il piano per raggiungere il deposito e si controlla che vi sia benzina a sufficienza per portare a compimento l'azione.

```
%vai al deposito
% azione che permette di raggiungere il deposito quando non e'
    possibile raggiungere alcun cassonetto
two_level_planner:add_del(0,va_da_a(P1,P2), St, [in(P2),
    serbatoio(Y)], [in(P1), serbatoio(X)], Cost) :-
   member(deposito(P2),St),
   member(in(P1), St),
   P1 \= P2,
   get_action_plan([in(P1), tager(P2)], va_da_a(P1,P2), _Plan,
       Cost),
% controllo che sia effettivamente possibile raggiungere il
    deposito con il livello di benzina attuale
   member(serbatoio(X), St),
   C is round(Cost),
   C @< X,
   Y is X - C.
```

3.2.5 RIFORNIMENTO

Quest'azione serve per rifornire il camion qualora l'agente si trovi presso il benzinaio. Rende vero il fluente che permette di riempire il serbatoio sino alla sua massima capienza.

```
% azione che permette di riempire il serbatoio del camion
two_level_planner:add_del(0,rifornimento, St, [serbatoio(C)],
    [serbatoio(X)], 0.5) :-
   member(serbatoio(X), St),
   member(in(P), St),
   member(benzinaio(P), St),
   capienza_serbatoio(C).
```

3.3 Funzione RACCOLTA

La funzione `raccolta` è stata modificata per adattarsi alle esigenze del nostro programma.

Forniamo di seguito la descrizione dei parametri della funzione `raccolta(W,P,Raccolta,Lista_aree,Lista_upperbound,B,Q,C,A)` :

- `W` il mondo in input
- L'agente si trova inizialmente in `P`

- Il deposito si trova in `Q`.
- Può fare rifornimento in `B`.
- `Lista_aree` lista contenente i punti che rappresentano le diverse distribuzioni di densità nel mondo
- `Lista_upperbound` lista che rappresenta gli upperbound corrispondenti ai punti contenuti in `Lista_aree`
- `C` è la capienza del serbatoio
- `A` è la capienza del camion

```
raccolta(W, Pos, Racc, Lista_aree, Lista_upperbound, B, Q, C, A) :-
% preparo lo stato iniziale del mondo+agente
   maplist(retractall, [in(_), deposito(_), deve_prendere(_),
       benzinaio(_), serbatoio(_), carico(_), cassonetto(_,_,_),
       capienza_camion(_),capienza_serbatoio(_), info_area(_,_)]),
   length(Lista_aree, La),
   length(Lista_upperbound, Lu),
   % le due liste devono essere lunghe uguali
   La == Lu,
   make_area(Lista_aree, Lista_upperbound),
   assert(in(Pos)),
   assert(deposito(Q)),
   assert(benzinaio(B)),
   assert(serbatoio(C)),
   assert(carico(0)),
   forall(member(P,Racc), assert(deve_prendere(P))),
   forall(member(P,Racc), inizializza_cassonetto(P, Lista_aree,
       Lista_upperbound)),
   assert(capienza_serbatoio(C)),
   assert(capienza_camion(A)),

%  disegno il mondo e il suo stato
   set_current_world(W),
   draw_world_state(W),

% fisso la strategia di default
   set_strategy(astar),
   set_strategy(ps(closed)),
   step(['Avvio con strategia astar e potatura chiusi']),!,

%  calcolo il piano di 2 livelli
   get_plan([Pos,Racc,Q,B], Plan, Cost),
   step(['Calcolato il piano ',Plan, '\ncon costo ', Cost]),!,

%  eseguo il piano di 2 livelli
   esegui(Plan).
```

Dopo aver ripulito la base di conoscenza dell'agente mediante la chiamata `retractall` per tutti i predicati necessari, controlliamo che la lista delle aree e la lista degli upperbound ad esse associati abbiano la stessa lunghezza.

`make_area` è una funzione ausiliaria utilizzata per inizializzare le aree dalle liste passate in input, la sua descrizione si trova nelle sezioni successive. Serve per poter inizializzare correttamente la quantità di spazzatura presente nei cassonetti in base all'area in cui si trovano attraverso la funzione `inizializza_cassonetto`.

Nella porzione successiva si scrivono nella KB tutte tutte le informazioni necessarie per rappresentare lo stato del mondo. In ultimo si scelgono le strategie, si calcola e si esegue il piano.

3.4 Funzioni ausiliarie

3.4.1 INIZIALIZZA_CASSONETTO

La funzione `inizializza_cassonetto` asserisce nella KB le informazioni sul cassonetto, ovvero il punto in cui si trova, l'area a cui appartiene e la quantità di spazzatura in esso presente. Per trovare la quantità di spazzatura consona è necessario identificare l'area di appartenenza del cassonetto mediante le funzioni `nearest` e `info_area`. La quantità di spazzatura viene inizializzata scegliendo un numero casuale tra 1 e l'upperbound.

```
pred(inizializza_cassonetto(point, list(point), list(integer))).
%  inizializza_cassonetto(P,Lista_aree, Lista_upperbound) assegna a
    un cassonetto nel punto P una quantita' di
%  rifiuti casuale dipendenti dall'area di appartenenza

inizializza_cassonetto(P, Lista_aree, Lista_upperbound) :-
% calcolo dell'area di appartenenza
   nearest(P, Lista_aree, Area),
%  generazione valore
   info_area(Area, Ub),
   random_between(1, Ub, R),
   assert(cassonetto(P,Area,R)).
```

3.4.2 MAKE_AREA e INFO_AREA

La funzione `make_area` prende in input la lista di punti rappresentanti le aree e quella dei loro upperbound. Successivamente, scorrendo le liste, rende vere nelle KB le informazioni relative alle aree con il loro upperbound definite da `info_area`.

```
pred(info_area(point,integer)).
:- dynamic(info_area/2).
```

```
pred(make_area(list(point),list(integer))).
%  date due liste inizializza le aree
%  MODO(++,++)det
make_area([Ha], [Hu]) :- assert(info_area(Ha,Hu)).
make_area([Ha|Ta], [Hu|Tu]) :- assert(info_area(Ha,Hu)) ,
    make_area(Ta, Tu).
```

3.4.3 NEAREST e LONGEST

Data una lista di punti e un punto P, queste funzioni hanno lo scopo di trovare il punto più vicino a P nella lista o quello più lontano.

```
pred(longest(point,list(point), point)).
% MODO(++,++,--)det
% longest(P,L,N) : N e' il punto appartente alla lista L piu'
    lontano da P
longest(P,[X], X).
longest(P, [X,Y|T], N) :-
     distance(diagonal,X,P,Dx),
     distance(diagonal,Y,P,Dy),
     Dx @> Dy, !,
     longest(P,[X|T], N).

longest(P, [X,Y|T], N) :-
     longest(P,[Y|T], N).


pred(nearest(point,list(point), point)).
% MODO(++,++,--)det
% nearest(P,L,N) : N e' il punto appartente alla lista L piu'
    vicino a P
nearest(P,[X], X).
nearest(P, [X,Y|T], N) :-
     distance(diagonal,X,P,Dx),
     distance(diagonal,Y,P,Dy),
     Dx @< Dy, !,
     nearest(P,[X|T], N).

nearest(P, [X,Y|T], N) :-
     nearest(P,[Y|T], N).
```

4 Euristiche

Nel codice sorgente del nostro progetto abbiamo inserito tre euristiche, due delle quali sottostimate che non permettono computazioni con un numero di cassonetti elevato ed una sovrastimata che in tempi assai brevi riesce

a ottenere piani con input molto maggiori rispetto al caso precedente ma soluzioni subottime. Procederemo fornendo la descrizione di tali euristiche e un'analisi statistica per mostrare di quanto le soluzioni ottenute utilizzando l'euristica efficiente ma sovrastimata distino dalle soluzioni ottime.

4.1 Euristica ES_lenta

```
two_level_planner:h(0, St, H) :-
   setof(X, member(deve_prendere(X), St), Pos) ->
      in(P),
      deposito(D),
      longest(P,Pos,L),
      distance(diagonal,P,L, Andata),
      distance(diagonal,L,D, Ritorno),
      H is Ritorno + Andata
      ;
      H = 0.
```

L'euristica viene calcolata sommando la distanza dal punto di partenza al cassonetto più lontano con la distanza tra il cassonetto più lontano e il deposito. Certamente quest'euristica é una sottostima poiché tiene conto solamente del tragitto per arrivare al cassonetto più lontano dal punto di partenza dell'agente e quello per raggiungere il deposito da tale cassonetto.

Infatti, nel caso in cui l'input sia solo un cassonetto e non vi sia bisogno di fare benzina l'euristica risulta essere sottostimata poiché per fare la stima si utilizza la distanza diagonale tra i punti da raggiungere, che è quella minore possibile. Da questa considerazione si evince che l'agente nel caso migliore percorre la stessa distanza stimata. In tutti gli altri casi è evidente che l'euristica risulti essere una sottostima.

4.2 Euristica ES_veloce

```
two_level_planner:h(0, St, H) :-
   setof(X, member(deve_prendere(X), St), Pos) ->
      in(P),
      deposito(D),
      p_merge_sort(P,Pos, Ord),
      add(Ord, D, Punti),
      distanza_es(Punti ,P, H)
      ;
      H = 0.
```

L'euristica viene calcolata trovando il cammino minimo per raggiungere tutti i cassonetti e il benzinaio partendo dal punto di partenza e terminando nel deposito.

Tale euristica risulta essere una sottostima più raffinata della precedente dal momento che si calcola il cammino minimo che congiunge tutti i punti ordinati in base alla distanza dall'agente. Nello specifico la funzione `p_merge_sort` si occupa di ordinare la lista dei cassonetti in funzione della distanza dal punto in cui l'agente si trova. Per vedere l'implementazione di tale funzione si guardi il codice sorgente. La funzione `distanza_es` non fa altro che calcolare la somma delle distanze che intercorrono tra un punto della lista e il successivo (si veda codice sorgente).

Questa è certamente una sottostima poiché consideriamo il cammino minimo per raggiungere tutti i punti senza considerare i possibili rifornimenti e la necessità di scaricare il camion.

4.3 Euristica ENS

```
two_level_planner:h(0, St, H) :-
      setof(X, member(deve_prendere(X), St), Pos) ->
      deposito(D),
      in(P),
      add(Pos,P, Res),
      distanza_ens(Res,D, V),
      H is V * 2
      ;
      H = 0.
```

L'euristica somma alla distanza tra il punto di partenza e il deposito la distanza tra il deposito e ogni cassonetto attraverso la funzione `distanza_ens` (si veda codice sorgente). La distanza tra ogni cassonetto e il deposito viene poi moltiplicata per 2 per simulare un tragitto di andata e ritorno.

L'euristica risulta in molti casi una sovrastima ma è risulta anche essere particolarmente efficiente come riscontrato in fase di testing per input sufficientemente grandi, dove il branching factor si attesta su 1,1.

5 Analisi Euristiche

Sono state condotte tre serie di test diversi per le tre euristiche presentate in cui vengono misurati il costo del piano trovato, il branching factor e il tempo di esecuzione per trovare il piano (tutti i test sono stati eseguiti sulla stessa macchina). In ogni serie di test è stato fatto variare il numero di cassonetti presenti nel mondo e l'inizializzazione di ogni cassonetto è stata forzata ad avere una quantità di rifiuti pari a 3. Tale scelta non va a modificare l'area di appartenenza del cassonetto. Ne riportiamo i risultati ottenuti.

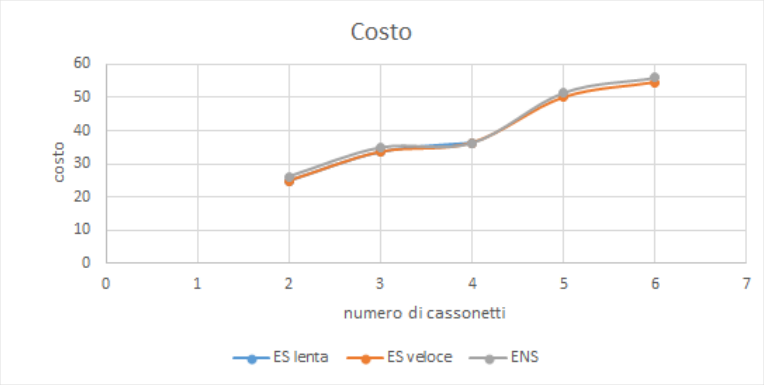
5.1 Prima serie

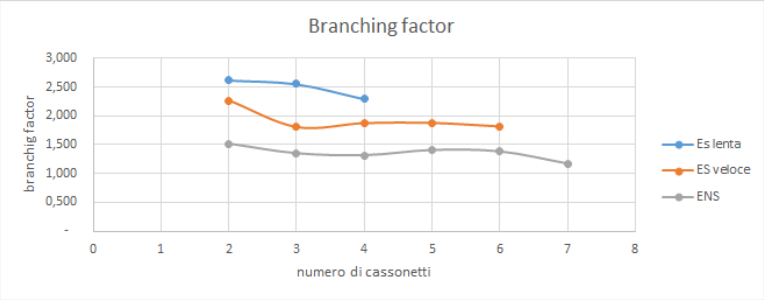
Per questa prima serie di test abbiamo utilizzato il seguente input, dove al numero di cassonetti n corrispondono i primi n punti della lista dei cassonetti:

```
raccolta(cw(1), point(2,2),[point(2,9), point(8,3), point(9,9),
    point(7,4), point(3,13), point(2,15), point(8,5)], [point(2,7),
    point(10,8)],[4,7],point(3,3),point(3,2),35,15).
```

Con questo input non è possibile fare ulteriori comparazioni poichè con tre punti ES_lenta va out of stack ed ES_veloce con sette.

Dal grafico sul costo presente nella pagina seguente si può notare come non vi siano sostanziali differenze tra le euristiche utilizzate in termini di costi ottenuti a fronte di velocità di calcolo molto differenti.





Nel grafico soprastante si può notare la differenza che sussiste nel branching factor delle varie euristiche alla profondità del nodo soluzione. L'euristica ENS risulta avere un branching factor molto minore rispetto alle altre due, e ciò consente di computare un numero molto maggiore di cassonetti rispetto a tutte le altre euristiche. Infatti, ENS è stata testata fino a 20 cassonetti, riportando un branching factor di circa 1,1.

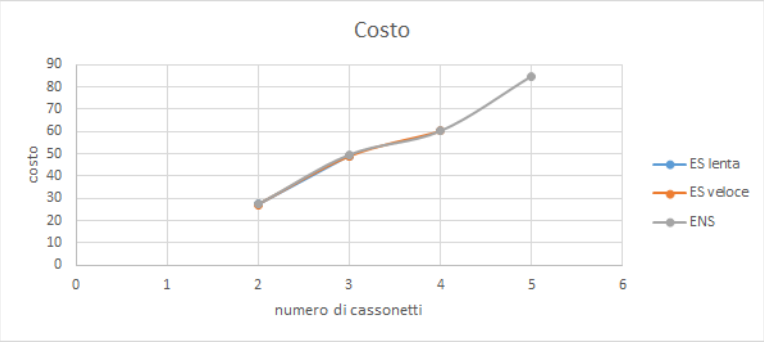
In questo terzo grafico si analizza il tempo di calcolo necessario per ottenere un piano sfruttando le varie euristiche. Come si nota entrambe le euristiche sottostimate mostrano un accentuato andamento esponenziale mentre l'euristica sovrastimata mantiene un comportamento sostanzialmente lineare con tempi di calcolo inferiori al secondo.

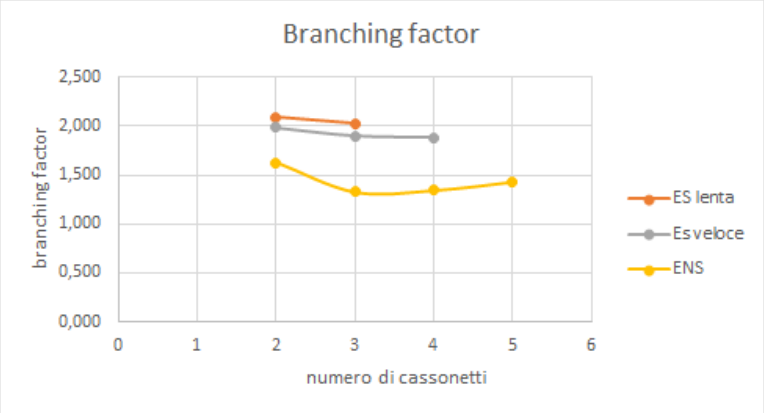
5.2 Seconda serie

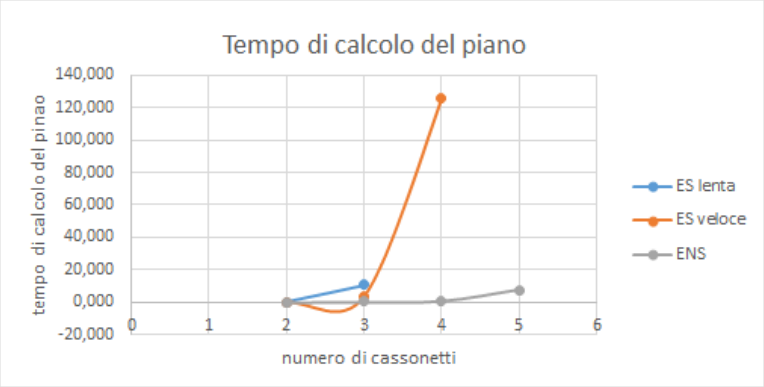
Questa seconda serie di test utilizziamo il seguente input che varia rispetto al precedente la capienza del camion, la quale viene settata ad un numero inferiore:

```
raccolta(cw(1), point(2,2),[point(2,9), point(8,3), point(9,9),
    point(7,4), point(3,13)], [point(2,7),
    point(10,8)],[3,3],point(3,3),point(3,2),35,3).
```

Anche in questo caso, sono valide le considerazioni ottenute nella casistica precedente. Alleghiamo in seguito i grafici per conferma e approfondimento.



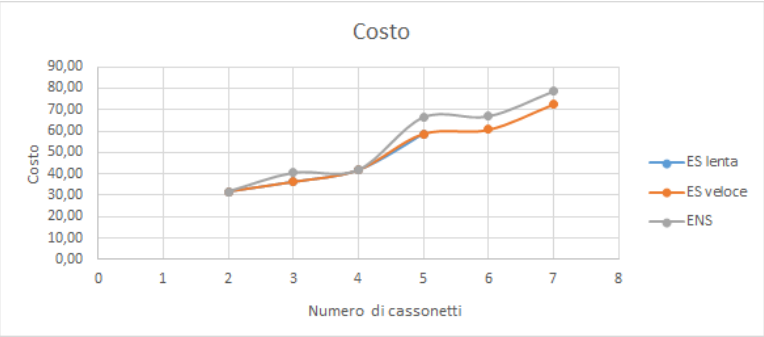




5.3 Terza serie

Per la terza serie di test è stato utilizzato il seguente input, che varia rispetto ai precedenti la posizione del deposito e della stazione di rifornimento (sono poste rispettivamente nell'angolo in alto a destra e in basso a destra) e la capienza del camion:

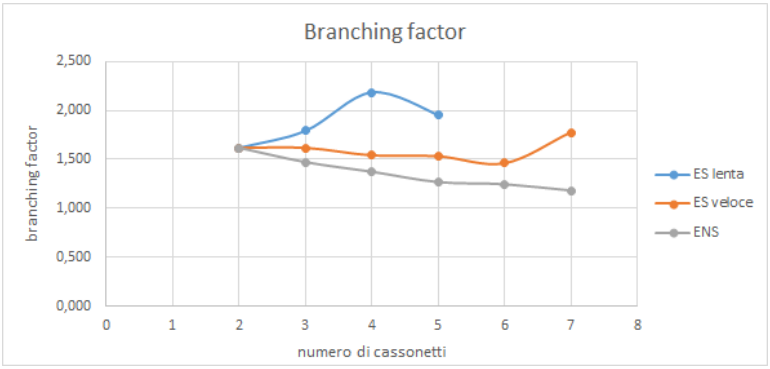
```
raccolta(cw(1), point(2,2),[point(2,9), point(8,3), point(9,9),
    point(7,4), point(3,13), point(2,15), point(8,5)], [point(2,7),
    point(10,8)],[4,7],point(2,19),point(9,19),50,15).
```

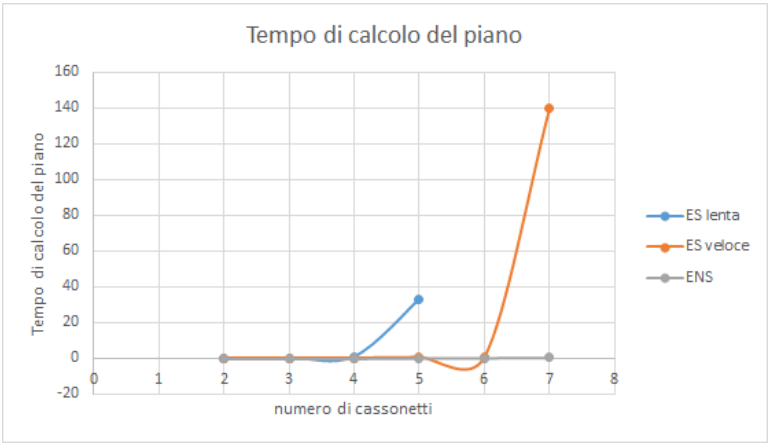


Per quanto riguarda costi, le due euristiche sottostimate si comportano in modo analogo, mentre l'euristica sovrastimata si discosta maggiormente dalle altre rispetto ai casi precedentemente trattati.

Un'altra considerazione degna di nota è che posizionando il deposito e la stazione di rifornimento più lontano dal punto di partenza dell'agente, le due euristiche sottostimate sono in grado di computare un numero maggiore di cassonetti, dal momento che l'albero di ricerca presenta molti più rami che falliscono in anticipo rispetto a quelli precedentemente trattati.

Per quanto riguarda le analisi sul Branching Factor e il tempo di calcolo, valgono le considerazioni precedenti. Ne riportiamo i grafici:





6 Considerazioni finali

In conclusione ci sembra una scelta accettabile utilizzare l'euristica sovrastimata poichè è incommensurabilmente più efficiente in tempo e spazio e non si discosta molto dalla soluzione ottima per quanto riguarda il costo del piano trovato.

NB. Nel caso in cui si volesse vedere più nel dettaglio i risultati dei test, si guardi il file test.txt.