« به نام او »

آموزش شبیه سازی دو بعدی فوتبال

نویسندگان : محمد علی میرزایی - علی یعقوبی

مرجع روبوکاپ ایران

www.iranrcss.com

جلسه دهم

۱- آموزش بیس **UVA_Trilearn**

۲- **آموزش هوش مصنوعی (A.I)، جست و جوی آگاهانه و اکتشاف**

در این جلسه قصد داریم به توضیح دفاع بپردازیم . در یک توضیح کلی، دفاع متشکل از کارهایی است برای جلوگیری از تحقق اهداف حریف و انواع گوناگون دارد که با توجه به حالت بازی متغیر است.

در هر تیم فوتبال کامل، سه قسمت اصلی وجود دارد :

۱ – بخش دفاع

۲ – بخش میانی

۳ – بخش حمله

در یک بازی هر سه بخش باید دست به دست هم دهند تا به پیروزی دست پیدا کنند ، اما تعریف پیروزی همیشه برد با نتیجه برتر نسبت به حریف نیست . گاهی ما اگر مساوی کنیم یا با اختلاف گل پایین بازی را واگذار کنیم، پیروزیم. زیرا ممکن است تیم حریف آنقدر قدرتمند باشد که نتوان در حال حاضر موفق به برد در برابر آن شد.

هرتیم برای دفاع کردن الگوریتم و روش مخصوص به خود را دارد . این بستگی به نوع بازی شما دارد که چگونه دفاع کنید . گاهی تیم قوی است و ما باید در لاک دفاعی فرو رویم و یا تیم بشدت ضعیف است و ما باید تا حد توان او را مورد فشار خط حمله قرار دهیم.

تا به اینجا ی آموزش (از جلسه اول تا به حال) یاد گرفتیم چگونه **واقعی** فکر کنیم، یعنی الگوریتم هایمان را بصورت منطقی پیاده سازی کنیم؛ حال که به این مرحله از الگوریتم پردازی و برنامه نویسی رسیده ایم، میتوانیم تفکرات خود را براحتی در تیم پیاده سازی کنیم.

به توضیح خط به خط تابع زیر می پردازیم :

```
۱ - SoccerCommand BasicPlayer::mark( ObjectT o, double dDist, MarkT mark )
۲ - {
۳ - VecPosition posMark = getMarkingPosition( o, dDist, mark );
۴ - VecPosition posAgent = WM->getAgentGlobalPosition();
۵ - VecPosition posBall = WM->getGlobalPosition( OBJECT_BALL );
۶ - // AngDeg angBody = WM->getAgentGlobalBodyAngle();
۷ - if( o == OBJECT_BALL )
۸ - {
۹ - if( posMark.getDistanceTo( posAgent ) < ۱.۵ )
۱۰ - return turnBodyToObject( OBJECT_BALL );
۱۱ - else
۱۲ - return moveToPos( posMark, ۳۱.۱, ۳.۱, false );
۱۳ - }
۱۴ - if( posAgent.getDistanceTo( posMark ) < ۲.۱ )
۱۵ - {
```

```
۱۶ - AngDeg angOpp = (WM->getGlobalPosition( o ) - posAgent).getDirection();

۱۷ - AngDeg angBall = (posBall - posAgent).getDirection();

۱۸ - if( isAngInInterval( angBall, angOpp,

۱۹ - VecPosition::normalizeAngle( angOpp + ۱۸۱ ) ) )

۲۰ - angOpp += ۸۱;

۲۱ - else

۲۲ - angOpp -= ۸۱;

۲۳ - angOpp = VecPosition::normalizeAngle( angOpp );

۲۴ - Log.log( ۵۱۳, "mark: turn body to ang %f", angOpp );

۲۵ - return turnBodyToPoint( posAgent + VecPosition( ۱.۱, angOpp, POLAR ) );

۲۶ - }

۲۷ - Log.log( ۵۱۳, "move to marking position" );

۲۸ - return moveToPos( posMark, ۲۵, ۳.۱, false );

۲۹ - }
```

در خط شماره ۳ تابعی به کار رفته است که در جلسه بعد توضیح خواهم داد . خط ۱ تا ۷ نیازی به توضیح ندارد . حال خط ۷ تا ۱۳ را مورد بررسی قرار می دهیم :

```
۷ - if( o == OBJECT_BALL )

۸ - {

۹ - if( posMark.getDistanceTo( posAgent ) < ۱.۵ )

۱۰ - return turnBodyToObject( OBJECT_BALL );

۱۱ - else

۱۲ - return moveToPos( posMark, ۳۱.۱, ۳.۱, false );

۱۳ - }
```

در خط شماره ۹ تابعی به نام getDistanceTo به کار رفته است . همانطور که از نام این تابع نیز مشخص است ، وظیفه این تابع یافتن فاصله ی بین دو VecPosition ای را که یکی به صورت اجرا کننده و یکی به صورت آرگومان به تابع داده شده است. در زیر توضیح اصلی موجود در بیس را مشاهده میکنیم :

```
/*! This method determines the distance between the current
    VecPosition and a given VecPosition. This is equal to the
    magnitude (length) of the vector connecting the two positions
    which is the difference vector between them.

    \param p a Vecposition
    \return the distance between the current VecPosition and the given
    VecPosition */
double VecPosition::getDistanceTo( const VecPosition p )
{
  return ( ( *this - p ).getMagnitude( ) );
}
```

* در جلسه بعد به توضیح تابع getMagnitude خواهیم پرداخت .

در این شرط بررسی می شود که اگر فاصله نقطه posMark (در جلسه بعد به طور مفصل توضیح خواهیم داد) تا خودمان کمتر از ۱.۵ است ، تنها بدن خود را بچرخاند و اگرنه به وسیله تابع زیر بازیکن را مجاب به تغییر مکان به نقطه PosMark می کنیم :

```
moveToPos( posMark, ۲۵, ۳.۱, false );
```

تابعی که در بالا معرفی شد ، یکی از اصلی ترین و پایه ای ترین توابع بیس می باشد . به توضیحات خود بیس در مورد این تابع توجه کنید :

```
/*! This skill enables an agent to move to a global position 'pos' on
    the field which is supplied to it as an argument. Since the agent
    can only move forwards or backwards into the direction of his
    body, the crucial decision in the execution of this skill is
    whether he should perform a turn or a dash. Turning has the
    advantage that in the next cycle the agent will be orientated
    correctly towards the point he wants to reach. However, it has the
    disadvantage that performing the turn will cost a cycle and will
    reduce the agent's velocity since no acceleration vector is added
    in that cycle.  Apart from the target position 'pos', this skill
    receives several additional arguments for determining whether a
    turn or dash should be performed in the current situation. If the
    target point is in front of the agent then a dash is performed
    when the relative angle to this point is smaller than a given
    angle 'angWhenToTurn'. However, if the target point is behind the
    agent then a dash is only performed if the distance to point is
    less than a given value 'dDistBack' and if the angle relative to
    the back direction of the agent is smaller than
    'angWhenToTurn'. In all other cases a turn is performed. Note that
    in the case of the goalkeeper it is sometimes desirable that he
    moves backwards towards his goal in order to keep sight of the
    rest of the field. To this end an additional boolean argument
    'bMoveBack' is supplied to this skill that indicates whether the
    agent should always move backwards to the target point. If this
    value equals true then the agent will turn his back towards the
    target point if the angle relative to his back direction is larger
```

```
    than 'angToTurn'. In all other cases he will perform a (backward)
    dash towards 'posTo' regardless of whether the distance to this
    point is larger than 'dDistBack'.

    \param posTo global target position to which the agent wants to move
    \param angWhenToTurn angle determining when turn command is returned
    \param dDistBack when posTo lies closer than this value to the back of
          the agent (and within angWhenToTurn) a backward dash is returned
    \param bMoveBack boolean determing whether to move backwards to 'posTo'
    \return SoccerCommand that determines next action to move to 'posTo' */
SoccerCommand BasicPlayer::moveToPos( VecPosition posTo, AngDeg angWhenToTurn,
                              double dDistBack, bool bMoveBack, int iCycles )
{
// previously we only turned relative to position in next cycle, now take
// angle relative to position when you're totally rolled out...
//  VecPosition posPred   = WM->predictAgentPos( 1, 0 );
  VecPosition posAgent  = WM->getAgentGlobalPosition();
  VecPosition posPred   = WM->predictFinalAgentPos();

  AngDeg      angBody   = WM->getAgentGlobalBodyAngle();
  AngDeg      angTo     = ( posTo - posPred ).getDirection();
              angTo     = VecPosition::normalizeAngle( angTo - angBody );
  AngDeg      angBackTo = VecPosition::normalizeAngle( angTo + 180 );

  double      dDist     = posAgent.getDistanceTo( posTo );

  Log.log( 509, "moveToPos (%f,%f): body %f to %f diff %f now %f when %f",
          posTo.getX(), posTo.getY(), angBody,
           ( posTo - posPred ).getDirection(), angTo,
           ( posTo -  WM->predictAgentPos( 1, 0 )).getDirection(),
           angWhenToTurn );
  if( bMoveBack )
  {
    if( fabs( angBackTo ) < angWhenToTurn )
      return dashToPoint( posTo, iCycles );
    else
      return turnBackToPoint( posTo );
  }
  else if(  fabs( angTo     ) < angWhenToTurn ||
           (fabs( angBackTo ) < angWhenToTurn && dDist < dDistBack ) )
    return dashToPoint( posTo, iCycles );
```

```
  else
    return directTowards( posTo, angWhenToTurn );
//return turnBodyToPoint( posTo );
}
```

ادامه مبحث "دفاع" را در جلسه آینده دنبال میکنیم.

جستجوی اکتشافی با حافظه محدود IDA*

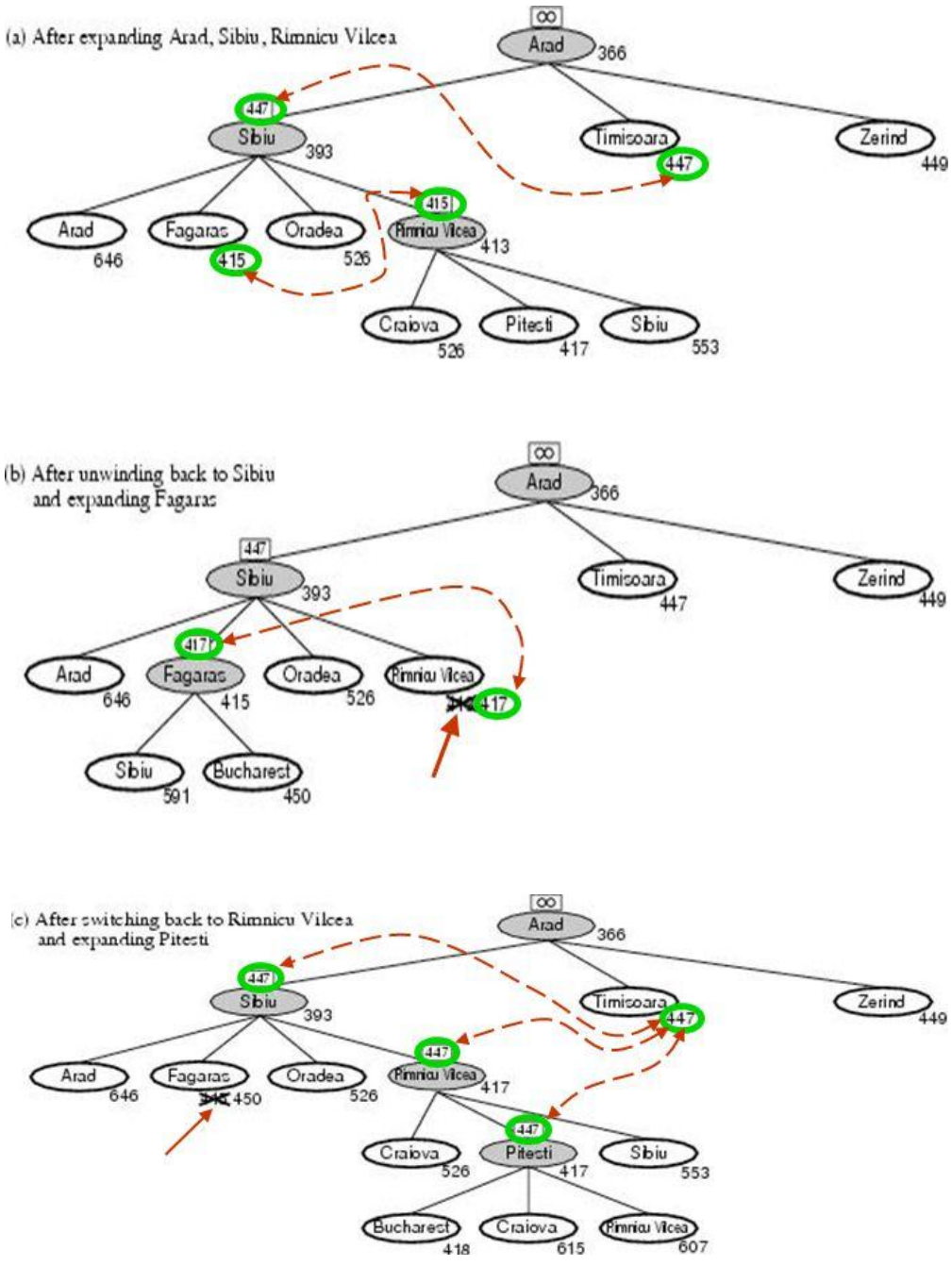
- ساده ترین راه برای کاهش حافظه مورد نیاز A* استفاده از عمیق کننده تکرار در زمینه جست و جوی اکتشافی است.
- الگوریتم عمیق کننده تکرار A* ⟵ IDA*
- در جستجوی IDA* مقدار برش مورد استفاده، عمق نیست بلکه هزینه f(g+h) است.
- IDA* برای اغلب مسئله های با هزینه های مرحله ای، مناسب است و از سربار ناشی از نگهداری صف مرتبی از گره ها اجتناب میکند.

بهترین جستجوی بازگشتی RBFS

- ساختار آن شبیه جست و جوی عمقی بازگشتی است، اما به جای اینکه دائما به طرف پایین مسیر حرکت کند، مقدار f مربوط به بهترین مسیر از هر جد گره فعلی را نگهداری میکند، اگر گره فعلی از این حد تجاوز کند، بازگشتی به عقب برمیگردد تا مسیر دیگری را انتخاب کند.
- این جستجو اگر تابع اکتشافی قابل قبولی داشته باشد، بهینه است.
- پیچیدگی فضایی آن O(bd) است.
- تعیین پیچیدگی زمانی آن به دقت تابع اکتشافی و میزان تغییر بهترین مسیر در اثر بسط گره ها بستگی دارد.
- RBFS تا حدی از IDA* کارآمدتر است، اما گره های زیادی تولید میکند.
- IDA* و RBFS در معرض افزایش توانی پیچیدگی قرار دارند که در جست و جوی گرافها مرسوم است، زیرا نمیتوانند حالتهای تکراری را در غیر از مسیر فعلی بررسی کنند. لذا، ممکن است یک حالت را چندین بار بررسی کنند.

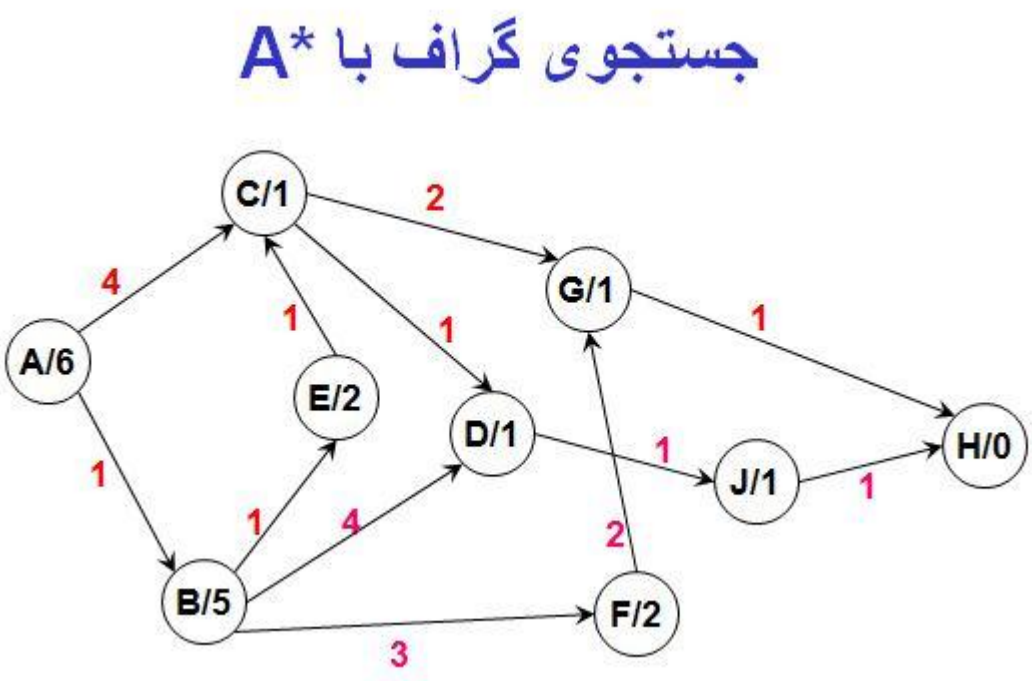
- IDA* و RBFS از فضای اندکی استفاده میکنند که به آنها آسیب میرساند. IDA* بین هر تکرار فقط یک عدد را نگهداری میکند که فعلی هزینه f است. RBFS اطلاعات بیشتری در حافظه نگهداری میکند.

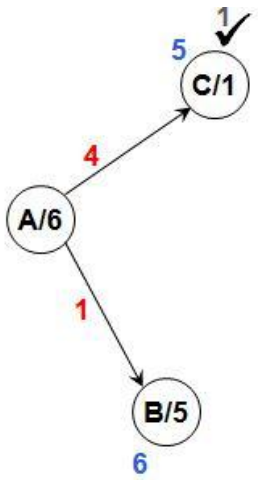
بهترین جستجوی بازگشتی در نقشه رومانی

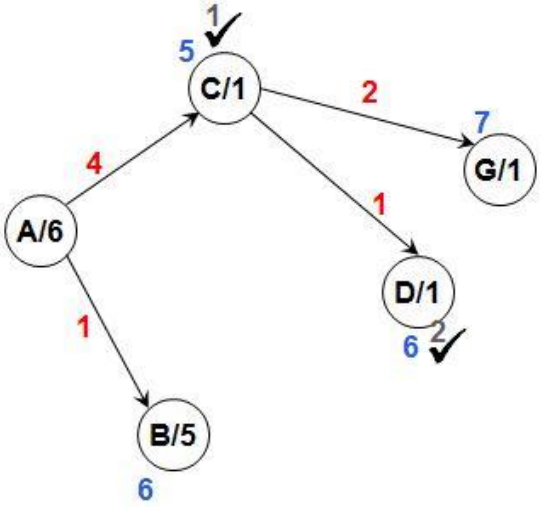


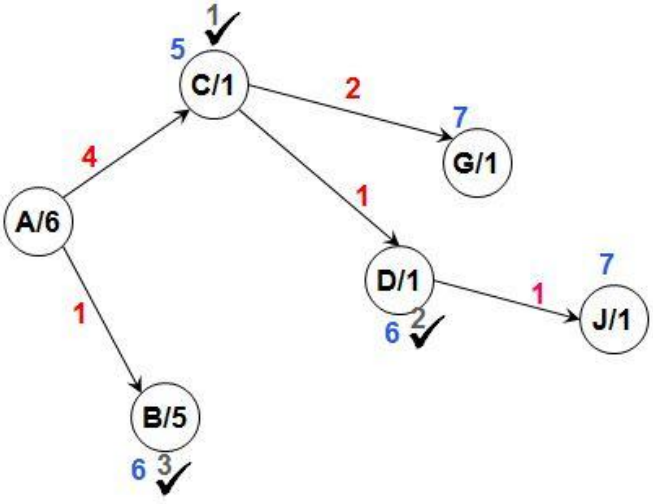
جستجوی حافظه محدود ساده SMA*

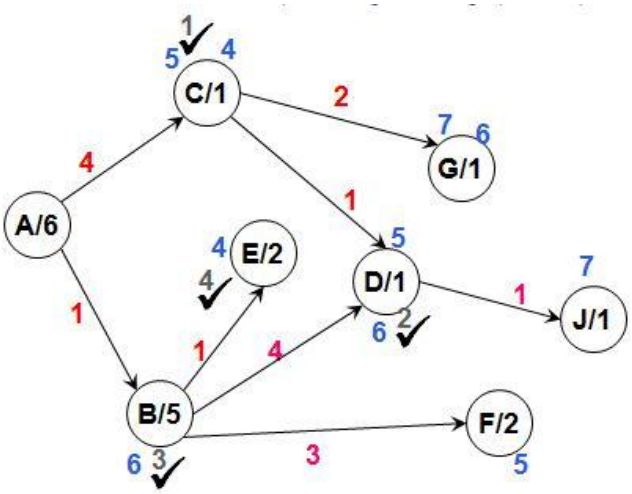
- SMA* بهترین برگ را بسط میدهد تا حافظه پر شود. در این نقطه بدون از بین بردن گره های قبلی نمیتواند گره جدیدی اضافه کند.
- SMA* همیشه بدترین گره برگ را حذف میکند و سپس از طریق گره فراموش شده به والد آن بر میگردد. پس جد زیر درخت فراموش شده، کیفیت بهترین مسیر را در آن زیر درخت میداند.
- اگر عمق سطحی ترین گره هدف کمتر از حافظه باشد, کامل است.
- SMA* بهترین الگوریتم همه منظوره برای یافتن حلهای بهینه میباشد.
- اگر مقدار f تمام برگها یکسان باشد و الگوریتم یک گره را هم برای بسط و هم برای حذف انتخاب کند، SMA* این مسئله را با بسط بهترین برگ جدید و حذف بهترین برگ قدیمی حل میکند.
- ممکن است SMA* مجبور شود دائما بین مجموعه ای از مسیرهای حل کاندید تغییر موضع دهد، در حالی که بخش کوچکی از هر کدام در حافظه جا شود.
- محدودیتهای حافظه ممکن است مسئله ها را از نظر زمان محاسباتی، غیر قابل حل کند.

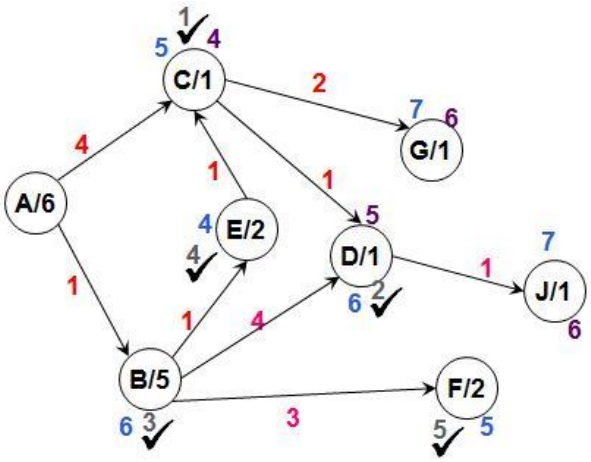


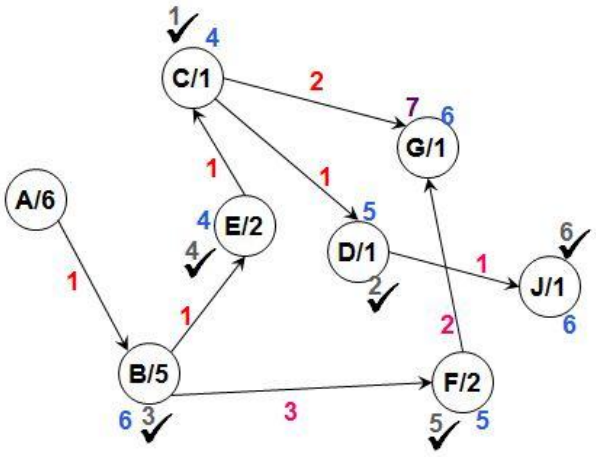

1
5
C/1
4
A/6
1
B/5
6

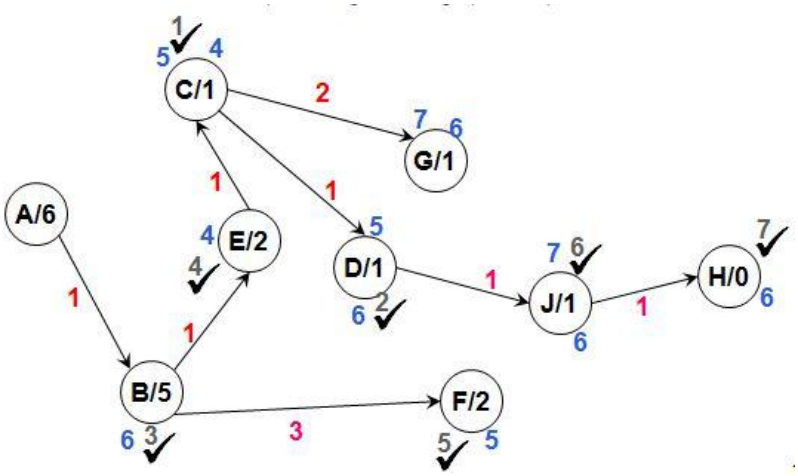

1
5
C/1
2
7
G/1
4
A/6
1
D/1
6
2
1
B/5
6


1
5
C/1
2
7
G/1
4
A/6
1
D/1
7
1
J/1
6
2
1
B/5
6
3


C/1
G/1
A/6
E/2
D/1
J/1
B/5
F/2


C/1
G/1
A/6
E/2
D/1
J/1
B/5
F/2


C/1
G/1
A/6
E/2
D/1
J/1
B/5
F/2



یادگیری برای جست و جوی بهتر

- روشهای جست و جوی قبلی، از **روشهای ثابت** استفاده میکردند.
- عامل با استفاده از **فضای حالت فراسطحی** میتواند یاد بگیرد که بهتر جست و جو کند.
- هر حالت در **فضای حالت فرا سطحی**، حالت(محاسباتی) داخلیِ برنامه ای را تسخیر میکند که فضای **حالت سطح شیء**، مثل رومانی را جست و جو میکند.
- الگوریتم **یادگیری فراسطحی** میتواند چیزهایی را از تجربیات بیاموزد تا زیردرختهای غیر قابل قبول را کاوش نکند.
- هدف یادگیری، **کمینه کردن کل هزینه**، حل مسئله است.

منبع :

Artificial Intelligence: A Modern Approach – by Stuart Russell and Peter Norvig