7. Паралелни алгоритми за графи и търсене

Васил Георгиев

 ci.fmi.uni-sofia.bg/

 v.georgiev@fmi.uni-sofia.bg

Съдържание

- Представяне и проблеми при графите
- Алгоритми за обхождане и път – последователни и паралелни версии
- Търсене в ширина, в дълбочина – последователни и паралелни версии
- Оптимиране с α-β minimax търсене

Графи – дефиниции

- графът $G(V, E)$ е двойка крайни множества на върховете (vertexes) и дъгите (edges); дъгата $e \in E$ е двойката $e=(u, v)$ $(u, v \in U)$, или наредената двойка $e=\langle u, v \rangle$ – насочен граф
- съседни (adjacent) са върховете, свързани с дъги (и в двата типа графи); N.B. $\exists \langle u, v \rangle \in E \Rightarrow u$ е съседен на v, не и обратното
- път в графа е последователност (наредено множество) от върхове $P=\{v_1, v_2, \ldots, v_k\}$: $\forall (v_i, v_{i+1}) \in E$, $1 \leq i \leq k$
- цикъл в граф: път $P=\{v_1, v_2, \ldots, v_k\}$: $v_1 \equiv v_k$; ациклични графи; прост цикъл
- пълен граф: всички върхове са съседни
- свързан граф: съществува път между всяка двойка върхове
- подграф: $G^l(V^l, E^l) \subseteq G(V, E)$: $V^l \subseteq V$ и $E^l \subseteq E$; свързан подграф (connected subgraph) – подграф на свързан граф, който запазва свойството свързаност между подмножеството върхове
- свързан компонент: свързан подграф G* (V, $E^l \subseteq E$) на ненасочения свързан граф G(V, E), за който $|E^l|$ = min (7.3)

Свойства в графите

- дърво: свързан ацикличен граф; с-ва:
  - $|E| = |V| - 1$
  - единствен път между всяка двойка върхове
- маркиран граф $G(V, E, W)$; $W(E) \rightarrow \Re$; тегло на граф и тегло на път (суми)
- двуделен граф: $\exists\ V_1$ и V_2: $V_1 \cap V_2 = \varnothing$ и $V_1 \cup V_2 = V$ и $\forall (u, v) \in E \Leftrightarrow (u \in V_k;\ v \in V - V_k)$ (възможна е бисекция)

Представяне на графите

- за алгоритмични цели графът $G(V, E)$ с $|V| = n$ се представя с матрици или списъци
- матрични форми:
  - матрица на съседство (adjacency matrix): $A_{n\times n}(a_{ij})$: a={1 $\Leftrightarrow$ (v_i, v_j)$\in E$ | 0}; за ненасочените графи A е симетрична (7.5.1)
  - тегловна матрица (weight matrix) за маркираните графи $W_{n\times n}(w_{ij})$: w_{ij} = {0 $\Leftrightarrow$ $i = j$ | $w(i, j) \Leftrightarrow (v_i, v_j)\in E$ | ∞};
  - матрица на свързност ({connectivity | reflexive | transitive closure} matrix) $C_{n\times n}(c_{ij})$: c_{ij} = {1 $\Leftrightarrow$ съществува ацикличен път P_{ij} | 0};
- списъчна форма: G се представя с n линейни свързани списъка със съседите на всеки връх – 7.5.2

Проблеми върху графи (с паралелно решение)

- обхождане
- минимално покриващо дърво (minimum spanning tree)
- най-къс път
- откриване на циклите
- задачата за търговския пътник

Обхождане

- прилагат се 3 базови алгоритъма, които имат паралелни версии:
  - търсене – “в дълбочина” (depth-first) или “в широчина” (breadth-first): избира се произволен възел v и всички негови съседи се маркират с v; в следващите стъпки итеративно се избира произволен немаркиран възел w и се повтаря първата стъпка за маркиране на съседите; сложност O(n + m) където |V| = n и |E| = m
  - с използване на матрицата на свързване
  - с разделяне на подграфи

Обхождане чрез матрицата на съседство (transitive closure)

- методът построява матрицата на свързване на графа $C_{n\times n}$, като степенува логически матрицата на съседство $A_{n\times n}$
- по принцип логическото умножение на булеви матрици (каквито са $A_{n\times n}$ и $C_{n\times n}$) представлява операцията (с *логическо* умножение и събиране!)

$$A_{n\times n} \times B_{n\times n} = C_{n\times n} : c_{ij} = (a_{i1}b_{1j}) + (a_{i2}b_{2j}) + \ldots + (a_{in}b_{nj})$$

- първата стъпка на метода е построяване на спомагателна матрица $B_{n\times n}$ която се получава от $A_{n\times n}$ с разполагане на 1 по главния диагонал; тогава $\forall b_{ik} \in B$ = {1 (ако има ацикличен път с дължина 0 или 1 от v_i до v_k) | 0 }
- следващите стъпки са итеративно намиране на продукта B^m ($m<n$), елементите на който отразяват съществуването на ацикличен път между съответните върхове с дължина не по-голяма от m; – пример 7.8

...Обхождане чрез матрицата на съседство

- всеки ацикличен път между два произволни върха на G е не по-дълъг от броя върхове $n \Rightarrow C = B^{n-1}$; на практика итеративните изчисления са $B, B^2, B^4, \ldots, B^m, C$, където $m = (n-1)/2$ алгоритъмът има $\lceil \text{lb}(n-1) \rceil$ итерации от матрични умножения
- когато размера на графа не е по степените на 2, m е най-малката степен на 2, по-голяма от $(n-1)$ – напр. за $A_{7\times7}$ $C = B^8$
- следващата стъпка е получаване на матрицата на свързващите компоненти $D_{n\times n}$ от C, където $\forall d_{jk} \in D = \{v_k$ (ако $c_{jk}=1) \mid 0\}$ $(0 \le j, k \le n-1)$ – ред j на D съдържа върховете, към които v_k образува свързан компонент с индекс x, където x минималния индекс на ненулев d_{jx}
- този метод е удобен за паралелна обработка тъй като се свежда до матрични изчисления с паралелизъм по данни; реда на изчисление е $A \rightarrow B \rightarrow C \rightarrow D$ в четири последователни цикъла за паралелни итерации (последния е за намиране на индекса на компонентите x)
- сложността на умножението на логически матрици е $\text{lb}(n-1)$ итерации – всяка с оценка $O(\text{lb}n)$, което дава обща сложност на алгоритъма $O(\text{lb}^2 n)$

Обхождане чрез разделяне (adjacency matrix partitioning)

- методът се състои в разеляне на матрицата на съседство $A_{n \times n}$ по редове на p части – колкото са обработващите процесори – като процесор P_i обработва подграфа $G_i(V, E_i)$, който се състои от съответните върхове и дъги – 7.10
- обработката на съответния подграф е откриване на неговото покриващо дърво чрез търсене, след което покриващите дървета на подграфите се сливат по двойки
- сливането на две покриващи дървета S_1 и S_2 – които имат най-много (n-1) общи върхове – се извършва като за всяка дъга $(u, v) \in S_1$ се проверява дали върховете й присъстват в S_2 – ако да – S_1 и S_2 се сливат в тези върхове, в противен случай се минава към следващата дъга на S_1
- алгоритъмът се състои главно в стъпка 1: локално търсене на покриващите дървета на подграфите и след това стъпка 2: сливане по двойки

...Обхождане чрез разделяне

- сложността на стъпка 1 за търсене на покриващото дърво в матрицата на съседство $A_{i\ (n/p)\times n}$ на подграфа G_i е $O(n^2/p)$
- сложността на стъпка 2 за сливане на покриващите дървета по двойки се състои от lb$p$ сливания с $O(n)$ сложност на всяко от тях, така че общата сложност на тази стъпка е $O(n \text{ lb}p)$
- общата сложност на паралелната версия на алгоритъма е $O((n^2/p)+(n \text{ lb}p))$

Път в маркиран граф

- маркиран граф $G(V, E, W)$; $W(E) \rightarrow \Re$; тегло на път $W(p) = W(v_1, v_2, \ldots, v_k) = \sum_{i=1, k-1} W(v_i, v_{i+1})$; най-къс път
- проблеми:
  - най къс път за двойка върхове
  - най-къс път за направление $d \in E$ от останалите върхове
  - най къс път с начало $s \in E$ до останалите върхове
  - най-къс път между всички двойки върхове – матрица D с най късите пътища – свойство: най-късия път между двойка върхове съдържа най-късите пътища между вложените двойки върхове (“Optimality principle”)
- методи за построяване на D:
  - маркиращ алгоритъм на Dijkstra (greedy метод)
  - алгоритъм на Floyd (динамичен метод)

Маркиращ алгоритъм на Dijkstra

- базира се на временна и крайна двойна маркировка на върховете j според пътищата им до тях от дадено начало *s*:
  - етикет $d(j)$ = дължината на най-късия път (s, j), минаващ само през върхове с крайна маркировка
  - етикет $p(j)$ = предшестващия j връх в (s, j)
- алгоритъм:
  - стъпка 1: крайно маркиране на *s* :
    - $d(s) = 0$; $p(s) = \varnothing$
    - временно маркиране на останалите върхове j: $d(j) = \infty$; $p(s) = \varnothing$
  - стъпка 2: ако *k* е последния връх с крайна маркировка, достижимите от него върхове j с временна маркировка се маркират:
    - $d(j) = \min\{ d(j), d(k)+d_{kj} \}$
    - $p(j) = \{k \; ; \; (d(j) = d(k)+d_{kj}) \mid p(j)\}$
  - стъпка 3: маркировката на върха с най-малко $d(j)$ става крайна; ако има върхове с временна маркировка ⇨ стъпка 2;
  - край

Паралелна версия на маркиращ алгоритъм на Dijkstra

$V_p = \{s\}$
forall $v \in (V \setminus V_p)$ do
  if (s, v) exists
    then d(v) = w
    else d(v) = ∞
endfor
while ($V_p \neq V$) do
  select vertex u: d(u) = min{d(v) | $v \in (V \setminus V_p)$}
  $V_p = V_p \cup \{u\}$
  forall $v \in (V \setminus V_p)$ do
    $d(v) = \min\{d(v), d(u) + w_{uv}\}$
  endfor
endwhile

- $O(n^2)$ – и $O(n^3)$ ако се търсят най-късите за всички двойки възли

Алгоритъм на Floyd за най-къс път между всички върхове

- базира се на
  - Optimality principle: $k \in P_{ij}$ ($k, i, j \in V$), ако P_{ij} е най късия път, тогава и P_{ik} P_{kj} са съответнита най-къси пътища и
  - "триъгълната операция" $w_{ij}(k) = \min\{ w_{ij}, w_{ik} + w_{kj}\}$ $\forall i,j \neq k$
- за "триъгалната операция" се доказва, че ако се приложи върху всички стойности на тегловната матрица на графа k=1, 2, …, n, то всички стойности на получената матрица са равни на най-късите пътища
- алгоритъмът стартира с модификация на тегловната матрица $W^{(0)}$, в която

$$w_{ij}^{(0)} = \min\{w_{ij}, (i, j) \in E \mid \infty , (i, j) \notin E \mid 0, i = j \}$$

...Алгоритъм на Floyd за най-къс път между всички върхове

- в първата (от n) стъпка $w_{ij} = w_{ij}(1)$ – т.е. триъгълна операция спрямо връх 1 (ако пътя през връх 1 е по-къс от дъгата, той я заменя) – резултат $W^{(1)}$
- при втората стъпка триъгълната операция се прилага спрямо $W^{(1)}$ и възел 2: $w_{ij}(2)^{(2)} = \min\{ w_{ij}^{(1)}, w_{i2}^{(1)} + w_{2j}^{(1)}\}$ (т.е. най-къс път само през върхове 1 и 2 – ако съществува такъв) – резултат $W^{(2)}$
- рекурентно: $w_{ij}^{(k)} = \min\{w_{ij}, k = 0 \mid \min \{ w_{ij}^{(k-1)}, w_{ik}^{(k-1)} + w_{kj}^{(k-1)}\}, 0 < k \leq n\text{-}1 \}$
- най-късите пътища между всички върхове се намират след n-тата стъпка с резултат $W^{(n)}$
- на практика търсената $D = W^{(m)}$ където $m = \lceil \mathrm{lb}(n - 1) \rceil$ - т.е. за n=7 m=3 като матриците по степените на 2 се получават чрез стандартно умножение

Последователна версия на алгоритъма на Floyd

- състои се от *n* матрични итерации, всяка с n^2 проверки, така че сложността е кубична – т.е. същия резултата както *n* пъти изпълнение на алгоритъма на Dijkstra; на практика обаче тази програма е с по- бързо изпълнение на отделните цикли и като цяло с по-кратък код
- псевдокод:

```
array D[n, n], W[n, n]
for i, j = (1, 2, …, n) D[n, n] ← W[n, n]
for k = (1, 2, …, n)
  for i = (1, 2, …, n)
    for j = (1, 2, …, n)
      D[i, j] ← min{D[i, j], (D[i, k] + D[k, j]}
return D
```

Паралелна версия на алгоритъма на Floyd

- версия за изпълнение от $p = n^2$ процесора, [логически] свързани в двумерна мрежа като процесор Pij изчислява последователно $w_{ij}^{(k)}$ (от стойностите на $w_{ij}^{(k-1)}$, $w_{ik}^{(k-1)}$ и $w_{kj}^{(k-1)}$) за $0 < k \leq n\text{-}1$
- броят на последователните итерации е $\lceil \text{lb} n \rceil$
- псевдокод (междинните резултати T е необходимо да се получат предварително, за да се избегне конфликтно четене и запис с последния израз):

```
array D[n, n], W[n, n], T[n, n]
forall i, j = (1, 2, …, n) in parallel do D[n, n] ← W[n, n]
repeat ⌈lb n⌉ times
  forall i, j, x = (1, 2, …, n) in parallel do
     T[i, x, j] ← D[i, x] + D[x, j]
  forall i, j = (1, 2, …, n) in parallel do
     D[i, j] ← min{D[i, j], T[i, 1, j], T[i, 2, j], .., T[i, n, j]}
return D
```

Паралелна версия на алгоритъма на Floyd

- версия за изпълнение от $p = n^2$ процесора, [логически] свързани в двумерна мрежа като процесор Pij изчислява последователно $w_{ij}^{(k)}$ (от стойностите на $w_{ij}^{(k-1)}$, $w_{ik}^{(k-1)}$ и $w_{kj}^{(k-1)}$) за $0 < k \leq n-1$
- броят на последователните итерации е $\lceil \text{lb} n \rceil$
- псевдокод (междинните резултати T е необходимо да се получат предварително, за да се избетне конфликтно четене и запис с последния израз):

```
array D[n, n], W[n, n], T[n, n]
forall i, j = (1, 2, …, n) in parallel do D[n, n] ← W[n, n]
repeat ⌈lb n⌉ times
  forall i, j, x = (1, 2, …, n) in parallel do
     T[i, x, j] ← D[i, x] + D[x, j]
  forall i, j = (1, 2, …, n) in parallel do
     D[i, j] ← min{D[i, j], T[i, 1, j], T[i, 2, j], .., T[i, n, j]}
return D
```

...Паралелна версия на алгоритъма на Floyd

- на *k*тата итерация алгоритъмът проверява за нов път от *i* до *j*, минаващ през повече от 2^{k-1} върха и през по-малко от 2^k върха, който да е "по-къс" от текущия най-къс път, и ако такъв връх е *x*, дължината се записва в *T*
- в последния израз разделянето на контекста по процесори избягва конфликта между новите и старите стойности в *D*

Задачи за търсене

- задачите за търсене са много широк клас и произтичат от разнообразни приложни области – най-често с представяне на проблемната област в термини от теорията на графите – и само сравнително неголяма част от тези алгоритми могат да се обработят на последователни архитектури за приемливо време
- търсенето се осъществява в различни структури данни – в зависимост от приложната област – поради което съществен е метода на търсене, който пряко произтича от обработваната структура (най-често в графи и дървета)
- основни методи:
  - търсене с разделяне (divide and conquer)
  - търсене в дълбочина (depth-first search, DFS)
  - търсене в ширина (breadth-first search, BFS)
  - оптимално (хибридно, евристично) търсене (best-first search)
  - търсене с разклоняване (branch and bound, BB)
  - оптимиране – търсене на оптимуми (alpha-beta minimax search)

Търсене с разделяне в сортиран списък

- последователно търсене с двоично разделяне - псевдокод:

```
location(index low, index high, x)                    {
  middle = ⌊(low + high)/2⌋
  if (x == Elements[middle]) return middle
  else if (x < Elements[middle]) return location(low, middle - 1, x)
       else return location(middle + 1, high, x)   }
```

- последователно търсене с многократно разделяне, при което всяка подобласт се решава рекурсивно в общата последователност - псевдокод:

```
Procedure Divide&Conquer(Input, Output)
        Divide(Input, Input1, Input2, …, InputM)
        for i = 1, M do
          Divide&Conquer(Input_i, Output_i)
        endfor
        Combine(Output1, …, OutputM, Output)
End      Divide&Conquer
```

Паралелни версии на търсене с разделяне в сортиран списък

- при мултикомпютрите се изгражда логическо дърво от процеси/процесори, което съответства на дървото за търсене; напр. върху процесорен хиперкуб е небходима предварителна фаза на картиране (mapping) така че да се минимизират междупроцесните комуникации (по дължина респ. време)
- при мултипроцесорите наличието на обща памет улеснява достъпа към областите на подпроблемите

Пример за паралелно търсене с разделяне

- n мерен вектор от сортирани елементи $S = \{E_1, E_2, \ldots, E_n\}$ се претърсва за стойност x от p-процесорна архитектура с обща памет ($p < n$)
- S се разделя на подвектори и всеки процесор P_i обработва последователните елементи $\{E_{n(i-1)/p+1}, E_{n(i-1)/p+2}, \ldots, E_{in/p}\}$ като прочита x в CR режим и ако за P_k $E_{n(k-1)/p+1} \leq x \leq E_{kn/p}$ – тогава се търси локално j: $x = E_{n(k-1)/p+j}$, резултатът е output = $(k-1)n/p + j$
- псевдокод:

```
Procedure Parallel_Divide&Conquer(Input, Output)
        Divide(Input, Input1, Input2, …, Input_p)
        for i = 1, p do in parallel
          Parallel_Divide&Conquer(Input_i, Output_i)
        endfor
        Combine(Output1, …, Output_p, Output)
End      Parallel_Divide&Conquer
```

Търсене в дълбочина

- по същество това са алгоритми за обхождане на графи (подобно на алгоритмите за покриващо дърво) – проверява се дървовидна структура за дадена стойност на атрибут на някой/и от върховете (и евентуално неговата позиция)
- за целта графите се представят със списък на съседство
- специфично за търсенето в дълбочина е, че обхождането на списъците продължава до намиране на връх, чиито съседи (елементите от неговия списък на съседство) са били вече проверени; след това с връщане назад на минимално разстояние обхождането продължава в нова посока (неизследвания връх)
- пример: ако сме стартирали от връх v и сме регистрирали дъгата (v, w) където w е непосетен, в следващата стъпка стартираме рекурсивно с w
- обратно – ако в горния сценарий w е вече посетен, не се преминава към следваща рекурсия и v остава връха, по чиито дъги се търси друг непосетен връх – т.е. именно проверката на върховете е в нарастваща дълбочина на дървото
- за удобство се приема конвенцията търсенето в наследниците (съседите) на даден връх да става отляво надясно; всеки проверен връх получава етикет-индекс DFI с реда му в последователността от проверки (в рамките на графа – не само в съответния клон)

Търсене в дълбочина – процедура

- параметър на процедурата е свързан граф, зададен със списък на съседство и стартов възел *v*, а резултат – индексите на върховете му (стартирайки с *v* = 1) – 7.26
- псевдокод:

```
Procedure DepthFirst(A)
        mark every vertex unvisited
        i = 1
        DepthFirstSearch(v)
Procedure DepthFirstSearch(v)
        mark v i
        for (each w adjacent to v) do
          if (w unvisited) DepthFirstSearch(w)
            i++
          endif
        endfor
End      DepthFirstSearch
End      DepthFirst
```

- сложност на последователния алгоритъм: *n* индексирания и *m* проверки O(*n* + *m*) – мощностите на *V* и *E*

Паралелно търсене в дълбочина

- DFS алгоритъмът е последователен по природа, за паралелно търсене е необходимо да се изследва матрицата на съседство (вместо списъка на съседство) и се въвежда списък на немаркираните съседи – U(v) – подмножество на списъка на съседите на v, ако възел w бъде посетен и маркиран, той се изключва от U(v) (както и от всички списъци на немаркирани съседи, в които присъства)
- резултатът е във фóрмата на два списъка – на дъгите и на клоните – като списъка дъги ARC_LIST(v) е всъщност списък на съседните върхове на v, а

Търсене в ширина

- търсене в ширина стартира от начален възел (корен) и проверява всички върхове на разстояние една дъга от него, след това – на две дъги и т.н. до проверка на всички върхове – на практика се построява минималното покриващо дърво (в немаркиран граф/дърво пътя/клона се измерва в брой дъги)
- отново конвенцията за проверка е отляво надясно и всеки проверен връх получава етикет-индекс BFI с реда му в последователността от проверки (в рамките на графа – не само в съответното ниво-дистанция от корена) – резултата е дърво, маркирано с индексите BFI (7.28)
- процедурата се базира на образуване на опашка от проверените съседи на текущия корен, върховете в която след това стават корени за търсенето на следващото ниво:

Търсене в ширина – процедура

```
Procedure BreadthFirstSearch
        mark every vertex unvisited
        initialize Queue with start vertex v
        i = 1    /* BFI
        while (Queue not empty) do
          remove the top q of Queue  /* v in the beginning
          for each vertex w adjacent to q
            if (w unvisited) then
              mark w = i
              i++
              place w in the bottom of Queue
            endif          endfor          endwhile
End
```

- сложността на последователния BFS е като на DFS (но с по-добри възможности за паралелна имплементация): n индексирания и m проверки $O(n + m)$ – мощностите на V и E
- прилагат се два подхода за паралелно BFS:
  - търсене по върхове (vertex-by-vertex BFS, VPBFS)
  - търсене по нива (level-by-level BFS, LPBFS)

Паралелно търсене в ширина – вариант VPBFS

- отново се използва списъци на немаркираните съседи $U(i)$ $(1 \le i \le n)$, матричен списък на съседството $ALM_{n\times(n-1)}$ и вектор на валентността (end-marker vector) $EM_{1\times n}$

```
Procedure ParallelBreadthFirstSearch_VP(ALM, EM, U)
    mark all vertices "unvisited"
    v ← start vertex
    mark v "visited"
    instruct processor(i) where 1 ≤ i ≤ k   /* k-node
system
      for j = 1 to k do
        if (k*(j-1)+1) <= EM(v)
          delete v from U(ALM(v, k*(j-1)+1))
        endif          endfor          endinstruct
    initialize Queue with v
    while (Queue not empty) do
      extract v from Queue
      for each w∈U(v) do
        mark w visited
        instruct processor(i) where 1 ≤ i ≤ k
          for j = 1 to k do
            if (k*(j-1)+1) <= EM(v) then
              delete w from U(ALM(w, k*(j-1)+1))
            endif              endfor              endinstruct
        add w to Queue
      endfor          endwhile
End
```

Оптимиране

- това е клас от задачи за търсене на оптимум на дискретни функции напр. при търсене на най-добър ход в игра (с противник)
- прилага се метода на α-β minimax търсене с изчерпателна проверка на всички възможни ходове (представени с върхове в дърво) и използване на връщане-назад; метода е от класа на ограниченото търсене в дълбочина с функция-критерий, която оценява възможните наследници на текущия връх
- най-добрата от оценките на наследниците се присвоява на родителя, избира се съответния наследник и респективно се оценяват неговите наследници (максимизиране) – 7.31.1
- търсенето в сценария на игра (неизвестен ход на противника) трябва да отчита не само оптималния ход (т.е. достижимо състояние) на фазата максимизиране, но и оценките на следващите достижими състояния, поради което при □-□ minimax към родителя се предават също “най-малко лошите” оценки на различните наследници от всяка негова дъга (фаза минимизиране) – (7.31.2)

α-β minimax оптимиращата стратегия

- α-β minimax оптимиращата стратегия е рекурсия със следните атрибути:
  - генератор на ходове – функция, която връща списък на достижимтие състояния за всеки играч
  - играч – може да бъде в позиция maximizer («играч») или minimizer («противник»)
  - функция-критерий – стойностите й се наричат статични оценки
  - критерий за край – индикатор за край (пределна дълбочина) на рекурсията – при който се избира (по функцията-критерий) ортималния ход при върха-родител; такива критерии обикновено са пределна дълбочина в брой нива или «игрално време»

α-β евристичен критерий

- при α-β minimax оптимирането се прилага евристична функция-критерий (α-β pruning) за ограничаване на търсенето
- стойността α е долната граница на оценката, която може да получи върха без да бъда отхвърлен; респ. β е горната граница (която може и да не бъде максималната оценка)
- правилата за контрол на търсенето чрез тези евристични параметри са:
  - търсенето не продължава след връх, за който играча (maximizer) получава α-оценка, не по-малка от β-оценката на противника (minimizer)
  - търсенето не продължава след връх, за който противника получава β-оценка, по-голяма от α-оценката на играча
- с помощта на такава функция се прочиства дървото на достижимите състояния от решения, които не могат да бъдат оптимални (но които съгласно “чистата” minimax стратегия подлежат на изследване) – т.е. на фазата максимизиране се премахват от разглеждане ходове, за които се установи че оценката ще е под текущия праг и на фаза минимизиране – при оценки над прага (пример 7.33)

Паралелно α-β minimax оптимиране

- този клас паралелни алгоритми обикновено има ниски стойности на насищане на ускорението (т.е. ниско ниво на паралелизма)
- паралелната обработка се базира на следните подходи:
  - паралелна генерация на ходовете и изчисляване на статичните оценки
  - паралелно търсене (обхождане)
- паралелната генерация и статични оценки има сравнително по-ниска линейност от паралелното търсене
- при паралелното търсене се разделя дървото T на клонове – BFS подход; в някои версии за по-големи системи се допуска и разделяне на заданието по нива при което процесорите се организират в логическо дърво