Domain-Specific Abstractions and Performance Portability

&

Data Access Complexity: Revisiting the Red/Blue Pebble game

P. (Saday) Sadayappan

The Ohio State University

Acknowledgements

Collaborators

Gerald Baumgartner(LSU)
Albert Cohen (ENS Paris)
Jason Cong (UCLA)
Franz Franchetti (CMU)
Robert Harrison (Stony Brook)
So Hirata (U. Illinois)
Jarek Nieploha (PNNL)
Srini Parthasarathy (OSU)
Louis-Noel Pouchet (UCLA)
Russ Pitzer (OSU, Chem)
J. Ramanujam (LSU)
Fabrice Rastello (ENS Lyon)
Nasko Rountev (OSU)
Vivek Sarkar (Rice)

Ph.D. Students

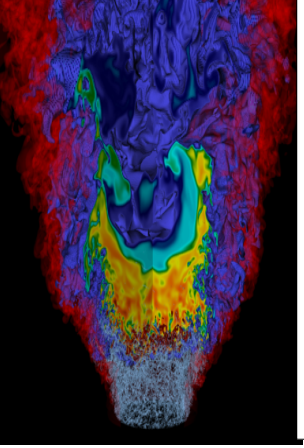
Muthu Baskaran
Uday Bondhugula
Jim Dinan
Xiaoyang Gao
Albert Hartono
Justin Holewinski
Sriram Krishnamoorthy
Qingda Lu
Mohammad Arafat
Venmugil Elango
Tom Henretty
Martin Kong
Pai-Wei Lai
Mahesh Ravishankar
Kevin Stock
Sanket Tavarageri

Funding

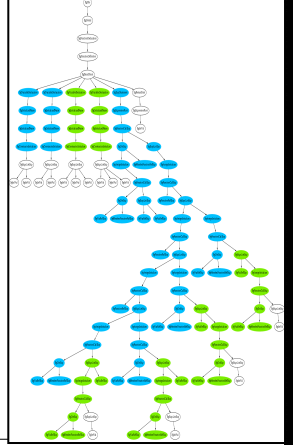
Natl. Science Foundn
Dept. of Energy
DARPA

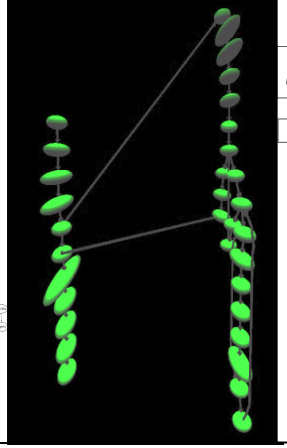
Why Domain-Specific Languages?

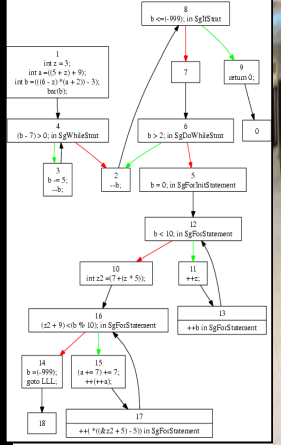
- Productivity
  - High level abstractions eases application development
- Performance
  - Domain-specific semantics enables specialized optimizations
  - Constraints on specification enables more effective general-purpose transformations and tuning (tiling, fusion)
- Portability
  - New architectures => changes only in domain-specific compiler, without any change in user application code

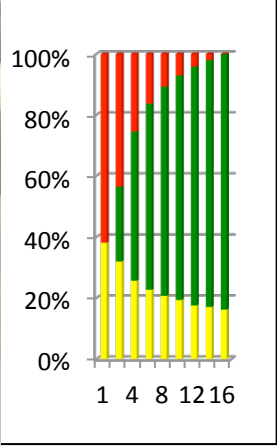


```
int aFunction(int a, int b)
{
      int c=b;
      return a;
}

main()
{
  int a,b,c,d,e;
  int i=4;
  for (i=0;i<10;i++)
  {
    int j=55;
    c=i+j;
    c=aFunction(i,c);
    a=aFunction(a+1,b);
  }
  #pragma SliceTarget
  a;
  return 0;
}
```









DSL Technology for Exascale Computing (D-TEC)

Lead PI
Daniel J. Quinlan
Lawrence Livermore National Laboratory

Deputy PI
Saman Amarasinghe
MIT

Co-PIs and Institutions
Armando Solar-Lezama, Adam Chlipala, Srinivas Devadas,
Una-May O'Reilly, Nir Shavit, Youssef Marzouk @ Massachusetts Institute of Technology
John Mellor-Crummey & Vivek Sarkar @ Rice University
Vijay Saraswat & David Grove @ IBM Watson
P. Sadayappan & Atanas Rountev @ Ohio State University
Ras Bodik @ University of California at Berkeley
Craig Rasmussen @ University of Oregon
Phil Colella @ Lawrence Berkeley National Laboratory
Rich Vuduc @ Georgia Tech
Scott Baden @ University of California at San Diego

The Tensor Contraction Engine
A Domain-Specific Compiler for Many-Body Methods in Quantum Chemistry

Oak Ridge National Laboratory
David E. Bernholdt, *Robert Harrison*

Pacific Northwest National Laboratory
Jarek Nieplocha

Louisiana State University
Gerald Baumgartner
J. Ramanujam

Ohio State University
Xiaoyang Gao, Albert Hartono, Sriram Krishnamoorthy, Qingda Lu, Alex Sibiryakov, ***Russell Pitzer, P. Sadayappan***

University of Florida
So Hirata

University of Waterloo
Marcel Nooijen

Supported by NSF and DOE

Time Crunch in Quantum Chemistry

Two major bottlenecks in computational chemistry:

- Very computationally intensive models
- Extremely time consuming to develop codes

The vicious cycle of computational science:

- More powerful computers make more accurate models computationally feasible :-)
- But efficient parallel implementation of complex models takes longer and longer
- Hence computational scientists spend more time with low-level programming for performance, and less time doing science :-(

- Coupled Cluster family of models in electronic structure theory
- Increasing number of terms => explosive increase in code complexity
- Theory is the same, but efficient implementations of higher order models took many years

Theory	#Terms	#F77Lines	Year
CCD	11	3209	1978
CCSD	48	13213	1982
CCSDT	102	33932	1988
CCSDTQ	183	79901	1992

CCSD Doubles Equation (Quantum Chemist's Eye Test Chart :-)

```
hbar[a,b,i,j] == sum[f[b,c]*t[i,j,a,c],{c}] -sum[f[k,c]*t[k,b]*t[i,j,a,c],{k,c}] +sum[f[a,c]*t[i,j,c,b],{c}] -sum[f[k,c]*t[k,a]*t[i,j,c,b],
    {k,c}] -sum[f[k,j]*t[i,k,a,b],{k}] -sum[f[k,c]*t[j,c]*t[i,k,a,b],{k,c}] -sum[f[k,i]*t[j,k,b,a],{k}] -sum[f[k,c]*t[i,c]*t[j,k,b,a],
    {k,c}] +sum[t[i,c]*t[j,d]*v[a,b,c,d],{c,d}] +sum[t[i,j,c,d]*v[a,b,c,d],{c,d}] +sum[t[j,c]*v[a,b,i,c],{c}] -sum[t[k,b]*v[a,k,i,j],
    {k}] +sum[t[i,c]*v[b,a,j,c],{c}] -sum[t[k,a]*v[b,k,j,i],{k}] -sum[t[k,d]*t[i,j,c,b]*v[k,a,c,d],{k,c,d}] -
    sum[t[i,c]*t[j,k,b,d]*v[k,a,c,d],{k,c,d}] -sum[t[j,c]*t[k,b]*v[k,a,c,i],{k,c}] +2*sum[t[j,k,b,c]*v[k,a,c,i],{k,c}] -
    sum[t[j,k,c,b]*v[k,a,c,i],{k,c}] -sum[t[i,c]*t[j,d]*t[k,b]*v[k,a,d,c],{k,c,d}] +2*sum[t[k,d]*t[i,j,c,b]*v[k,a,d,c],{k,c,d}] -
    sum[t[k,b]*t[i,j,c,d]*v[k,a,d,c],{k,c,d}] -sum[t[j,d]*t[i,k,c,b]*v[k,a,d,c],{k,c,d}] +2*sum[t[i,c]*t[j,k,b,d]*v[k,a,d,c],{k,c,d}] -
    sum[t[i,c]*t[j,k,d,b]*v[k,a,d,c],{k,c,d}] -sum[t[j,k,b,c]*v[k,a,i,c],{k,c}] -sum[t[i,c]*t[k,b]*v[k,a,j,c],{k,c}] -
    sum[t[i,k,c,b]*v[k,a,j,c],{k,c}] -sum[t[i,c]*t[j,d]*t[k,a]*v[k,b,c,d],{k,c,d}] -sum[t[k,d]*t[i,j,a,c]*v[k,b,c,d],{k,c,d}] -
    sum[t[k,a]*t[i,j,c,d]*v[k,b,c,d],{k,c,d}] +2*sum[t[j,d]*t[i,k,a,c]*v[k,b,c,d],{k,c,d}] -sum[t[j,d]*t[i,k,c,a]*v[k,b,c,d],{k,c,d}] -
    sum[t[i,c]*t[j,k,d,a]*v[k,b,c,d],{k,c,d}] -sum[t[i,c]*t[k,a]*v[k,b,c,j],{k,c}] +2*sum[t[i,k,a,c]*v[k,b,c,j],{k,c}] -
    sum[t[i,k,c,a]*v[k,b,c,j],{k,c}] +2*sum[t[k,d]*t[i,j,a,c]*v[k,b,d,c],{k,c,d}] -sum[t[j,d]*t[i,k,a,c]*v[k,b,d,c],{k,c,d}] -
    sum[t[j,c]*t[k,a]*v[k,b,i,c],{k,c}] -sum[t[j,k,c,a]*v[k,b,i,c],{k,c}] -sum[t[i,k,a,c]*v[k,b,j,c],{k,c}]
    +sum[t[i,c]*t[j,d]*t[k,a]*t[l,b]*v[k,l,c,d],{k,l,c,d}] -2*sum[t[k,b]*t[l,d]*t[i,j,a,c]*v[k,l,c,d],{k,l,c,d}]
    -2*sum[t[k,a]*t[l,d]*t[i,j,c,b]*v[k,l,c,d],{k,l,c,d}] +sum[t[k,a]*t[l,b]*t[i,j,c,d]*v[k,l,c,d],{k,l,c,d}]
    -2*sum[t[j,c]*t[l,d]*t[i,k,a,b]*v[k,l,c,d],{k,l,c,d}] -2*sum[t[j,d]*t[l,b]*t[i,k,a,c]*v[k,l,c,d],{k,l,c,d}]
    +sum[t[j,d]*t[l,b]*t[i,k,c,a]*v[k,l,c,d],{k,l,c,d}] -2*sum[t[i,c]*t[l,d]*t[j,k,b,a]*v[k,l,c,d],{k,l,c,d}]
    +sum[t[i,c]*t[l,a]*t[j,k,b,d]*v[k,l,c,d],{k,l,c,d}] +sum[t[i,c]*t[l,b]*t[j,k,d,a]*v[k,l,c,d],{k,l,c,d}]
    +sum[t[i,k,c,d]*t[j,l,b,a]*v[k,l,c,d],{k,l,c,d}] +4*sum[t[i,k,a,c]*t[j,l,b,d]*v[k,l,c,d],{k,l,c,d}]
    -2*sum[t[i,k,c,a]*t[j,l,b,d]*v[k,l,c,d],{k,l,c,d}] -2*sum[t[i,k,a,b]*t[j,l,c,d]*v[k,l,c,d],{k,l,c,d}]
    -2*sum[t[i,k,a,c]*t[j,l,d,b]*v[k,l,c,d],{k,l,c,d}] +sum[t[i,k,c,a]*t[j,l,d,b]*v[k,l,c,d],{k,l,c,d}]
    +sum[t[i,c]*t[j,d]*t[k,l,a,b]*v[k,l,c,d],{k,l,c,d}] +sum[t[i,j,c,d]*t[k,l,a,b]*v[k,l,c,d],{k,l,c,d}]
    -2*sum[t[i,j,c,b]*t[k,l,a,d]*v[k,l,c,d],{k,l,c,d}] -2*sum[t[i,j,a,c]*t[k,l,b,d]*v[k,l,c,d],{k,l,c,d}]
    +sum[t[j,c]*t[k,b]*t[l,a]*v[k,l,c,i],{k,l,c}] +sum[t[l,c]*t[j,k,b,a]*v[k,l,c,i],{k,l,c}] -2*sum[t[l,a]*t[j,k,b,c]*v[k,l,c,i],{k,l,c}]
    +sum[t[l,a]*t[j,k,c,b]*v[k,l,c,i],{k,l,c}] -2*sum[t[k,c]*t[j,l,b,a]*v[k,l,c,i],{k,l,c}] +sum[t[k,a]*t[j,l,b,c]*v[k,l,c,i],{k,l,c}]
    +sum[t[k,b]*t[j,l,c,a]*v[k,l,c,i],{k,l,c}] +sum[t[j,c]*t[l,k,a,b]*v[k,l,c,i],{k,l,c}] +sum[t[i,c]*t[k,a]*t[l,b]*v[k,l,c,j],{k,l,c}]
    +sum[t[l,c]*t[i,k,a,b]*v[k,l,c,j],{k,l,c}] -2*sum[t[l,b]*t[i,k,a,c]*v[k,l,c,j],{k,l,c}] +sum[t[l,b]*t[i,k,c,a]*v[k,l,c,j],{k,l,c}]
    +sum[t[i,c]*t[k,l,a,b]*v[k,l,c,j],{k,l,c}] +sum[t[j,c]*t[l,d]*t[i,k,a,b]*v[k,l,d,c],{k,l,c,d}] +sum[t[j,d]*t[l,b]*t[i,k,a,c]*v[k,l,d,c],
    {k,l,c,d}] +sum[t[j,d]*t[l,a]*t[i,k,c,b]*v[k,l,d,c],{k,l,c,d}] -2*sum[t[i,k,c,d]*t[j,l,b,a]*v[k,l,d,c],{k,l,c,d}]
    -2*sum[t[i,k,a,c]*t[j,l,b,d]*v[k,l,d,c],{k,l,c,d}] +sum[t[i,k,c,a]*t[j,l,b,d]*v[k,l,d,c],{k,l,c,d}]
    +sum[t[i,k,a,b]*t[j,l,c,d]*v[k,l,d,c],{k,l,c,d}] +sum[t[i,k,c,b]*t[j,l,d,a]*v[k,l,d,c],{k,l,c,d}] +sum[t[i,k,a,c]*t[j,l,d,b]*v[k,l,d,c],
    {k,l,c,d}] +sum[t[k,a]*t[l,b]*v[k,l,i,j],{k,l}] +sum[t[k,l,a,b]*v[k,l,i,j],{k,l}] +sum[t[k,b]*t[l,d]*t[i,j,a,c]*v[l,k,c,d],{k,l,c,d}]
```

Example: A CCSD Tensor Contraction

$$C_{ijkl} = \sum_{mn} A_{imkn} \cdot B_{jnlm}$$

```
for (i=0; i<P; i++)
 for (j=0; j<Q; j++)
  for (k=0; k<R; k++)
   for (l=0; l<S; l++)
    for (m=0; m<T; m++)
     for (n=0; n<U; n++)
      C[i][j][k][l] += A[i][m][k][n]*B[j][n][l][m];
```

- Tensor contraction is a generalized high-dimension analog of matrix-matrix product
- Each loop index appears in exactly two tensors
  - Contraction indices appear only in the input (r.h.s) tensors: m and n
  - External indices appear in output tensor and one input tensor: {i, k} and {j, l}
- Each loop is either a parallel loop or a reduction loop
- Relative order of the contraction indices in loop nest is unimportant (associative reordering of reduction)
- Loop nests can be considered fully permutable (dependence analysis in existing compilers will not permit permutation of contraction indices)

Tensor Contraction Engine

- Automatic transformation from high-level specification
  - Chemist specifies computation in high-level mathematical form
  - Synthesis system transforms it to efficient parallel program
  - Code is tailored to target machine
  - Code can be optimized for specific molecules being modeled
- Multi-institutional collaboration (OSU, LSU, Waterloo, ORNL, PNNL, U. Florida)
- Two versions of TCE developed
  - a) Full chemistry, but fewer optimizations (Hirata) b) Excluded some details, but sophisticated optimizations
  - Used to implement over 20 models, in latest release of NWChem (a few million lines of synthesized code)
  - First parallel implementation for many of the methods
  - New improved TCE-2 planned

$$A3A = \tfrac{1}{2}\left(X_{ce,af}Y_{ae,cf} + X_{\bar{c}\bar{e},\bar{a}\bar{f}}Y_{\bar{a}\bar{e},\bar{c}\bar{f}} + X_{\bar{c}\bar{e},\bar{a}f}Y_{\bar{a}\bar{e},cf} + X_{\bar{c}\bar{e},\bar{a}\bar{f}}Y_{\bar{a}\bar{e},\bar{c}\bar{f}} + X_{\bar{c}\bar{e},a\bar{f}}Y_{\bar{a}\bar{e},c\bar{f}} + X_{\bar{c}\bar{e},\bar{a}\bar{f}}Y_{\bar{a}\bar{e},\bar{c}\bar{f}}\right)$$

$$X_{ce,af} = t_{ij}^{ce}t_{ij}^{af} \qquad Y_{ae,cf} = \langle ab \| ek \rangle \langle cb \| fk \rangle$$

```
range V = 3000;
range O = 100;

index a,b,c,d,e,f : V;
index i,j,k : O;

mlimit = 10000000;

function F1(V,V,V,O);
function F2(V,V,V,O);

procedure P(in T1[O,O,V,V], in T2[O,O,V,V], out X)=

begin
   A3A == sum[ sum[F1(a,b,e,k) * F2(c,f,b,k), {b,k}]
           * sum[T1[i,j,c,e] * T2[i,j,a,f], {i,j}],
           {a,e,c,f}]*0.5 + ...;
end
```

Customized Kernel Generation for Tensor Contractions

- Effective SIMD utilization is increasingly important for high performance on current/ emerging processors
- Automatic vectorization by production compilers (even with manual unrolling) often results in performance well under 50% of machine peak
- Customized code generator (using vector intrinsics) for tensor contractions

Approach to Code Generation

Overall Approach

- Explicitly generate vector-intrinsics based code for TC
- Compile generated code using icc/gcc
- Use Machine Learning model to predict performance of generated assembly code
- Explore space of code variants and choose the one with highest predicted performance

Search Space

Vectorized Dimension Determines how memory is accessed

Loop Permutation Enable hoisting loads/stores outside inner loops

Unroll-and-Jam Enable register reuse

Example: Multi-resolution Kernel

Kernel

$$R_{ijk} = \sum_{i'j'k'} S_{i'j'k'} X_{i'i} Y_{j'j} Z_{k'k}$$

- Used extensively in MADNESS (Multi-resolution Adaptive Numerical Environment for Scientific Simulation)
- Tensors are small, frequently fitting completely within L1-cache

Low Rank Decomposition

$$X_{i'i} = \sum_{l} X^{L}_{il} X^{R}_{li'}$$

$$Y_{j'j} = \sum_{m} Y^{L}_{jm} Y^{R}_{mj'}$$

$$Z_{k'k} = \sum_{n} Z^{L}_{kn} Z^{R}_{nk'}$$

Example: Multi-resolution Kernel

Kernel

$$R_{ijk} = \sum_{i'j'k'} S_{i'j'k'} X_{i'i} Y_{j'j} Z_{k'k}$$

- Implemented as a series of six tensor contractions

$$A_{ljk} = \sum_{i} S_{ijk} \cdot X^{L}_{il}$$

$$B_{lmk} = \sum_{j} A_{ljk} \cdot Y^{L}_{jm}$$

$$C_{lmn} = \sum_{k} B_{lmk} \cdot Z^{L}_{kn}$$

$$D_{lmk'} = \sum_{n} C_{lmn} \cdot Z^{R}_{nk'}$$

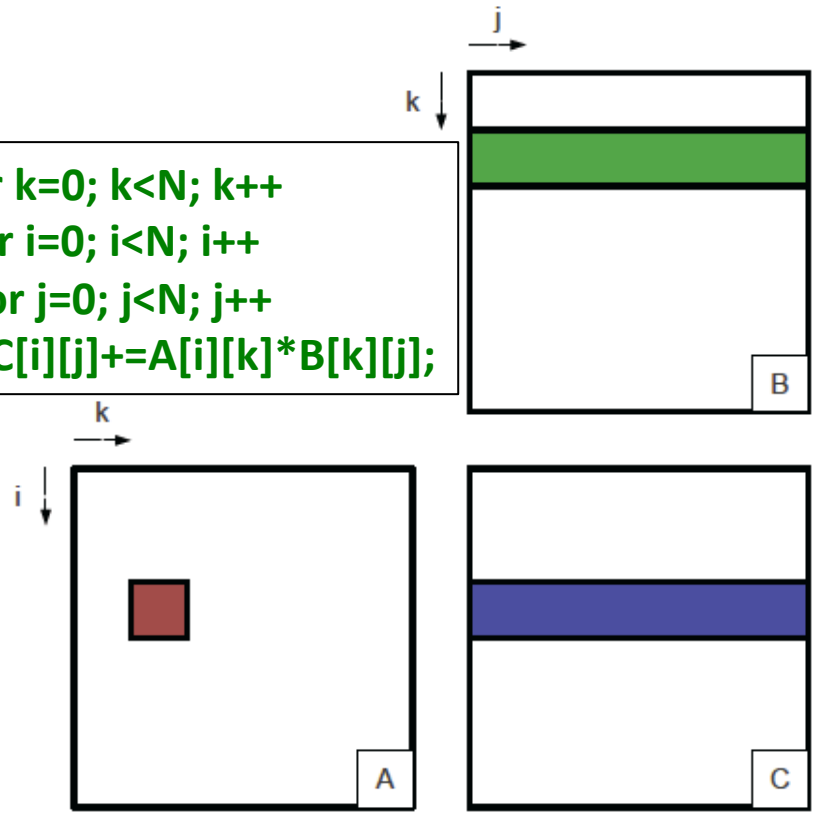
$$E_{lj'k'} = \sum_{m} D_{lmk'} \cdot Y^{R}_{mj'}$$

$$R_{i'j'k'} = \sum_{l} E_{lj'k'} \cdot X^{R}_{li'}$$

Vectorization Dimensions

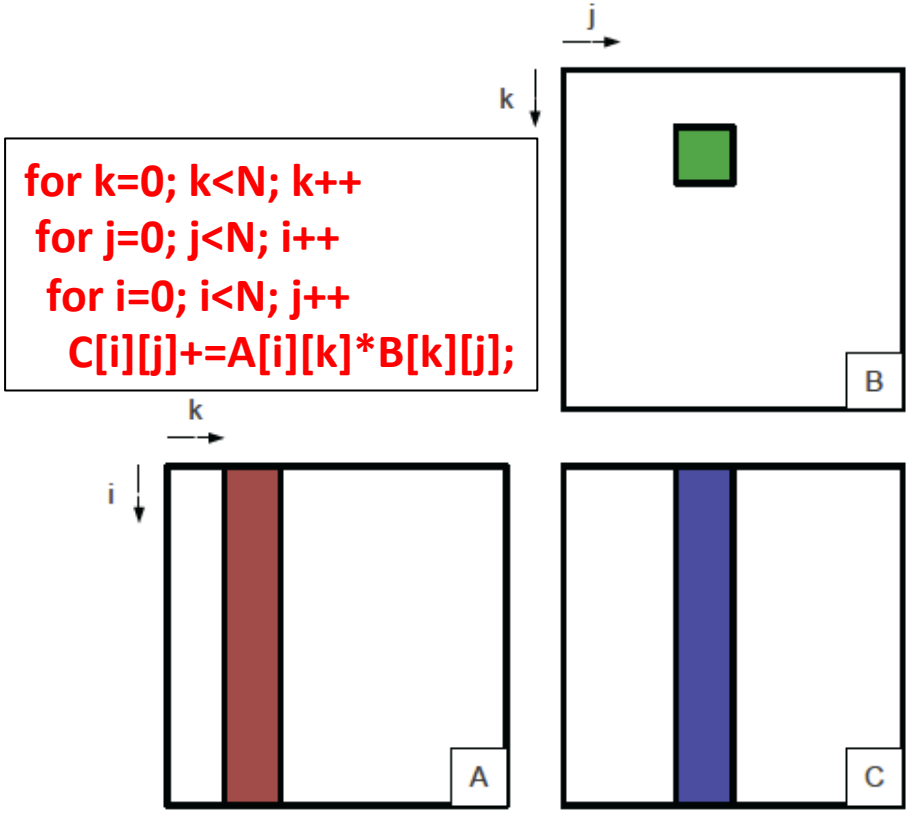
Inner loop "j" is good for Vectorization (stride is 0 or 1)

```
for k=0; k<N; k++
 for i=0; i<N; i++
  for j=0; j<N; j++
   C[i][j]+=A[i][k]*B[k][j];
```



Inner loop "i" is bad for vectorization (access stride is N)

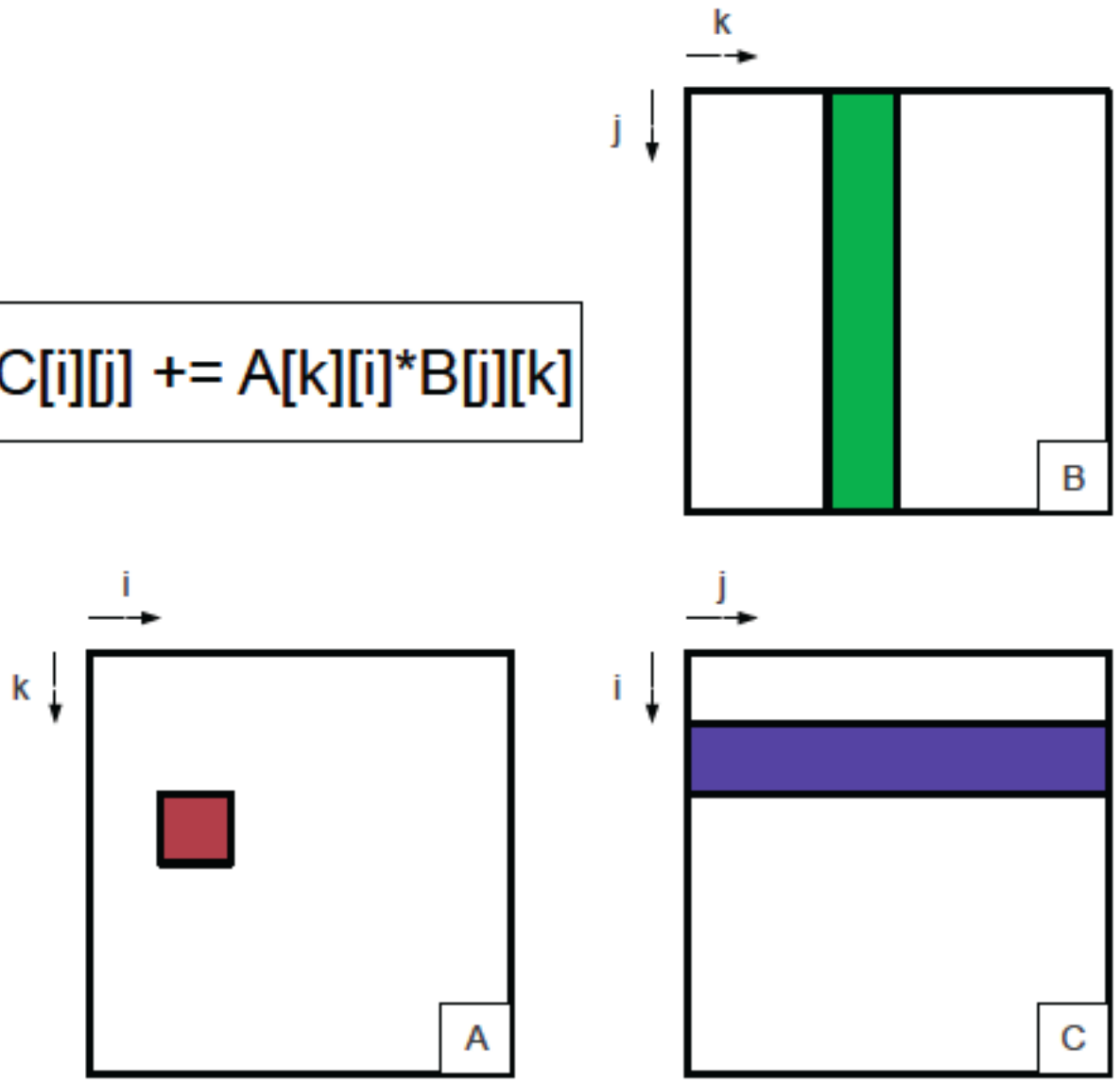
```
for k=0; k<N; k++
 for j=0; j<N; i++
  for i=0; i<N; j++
   C[i][j]+=A[i][k]*B[k][j];
```



What if No Vectorizable Loop?

```
for k=0; k<N; k++
 for j=0; j<N ;i++
  for i=0; i<N; j++
    C[i][j]+=A[k][i]*B[j][k];
```

C[i][j] += A[k][i]*B[j][k]

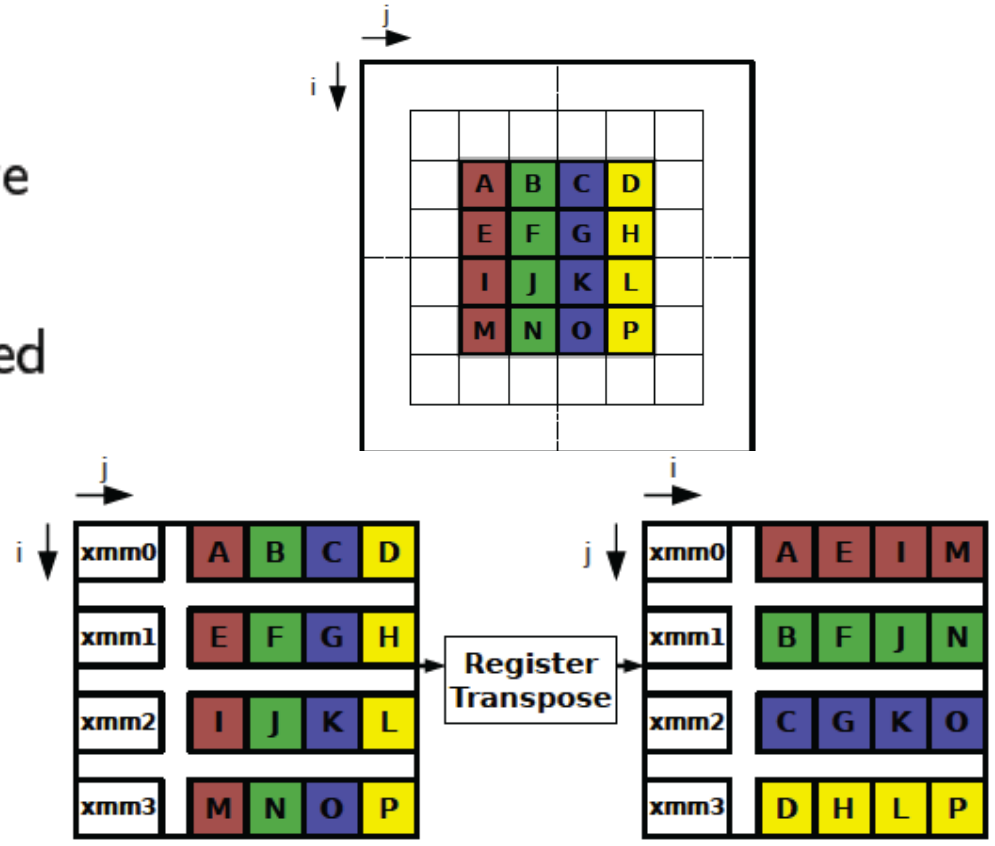


What if No Vectorizable Loop?

We can still vectorize by use of in-register transpose

Cost of register transpose often amortizable

- Vectors along columns are desired
- Vectors can only be loaded by rows



Code Example

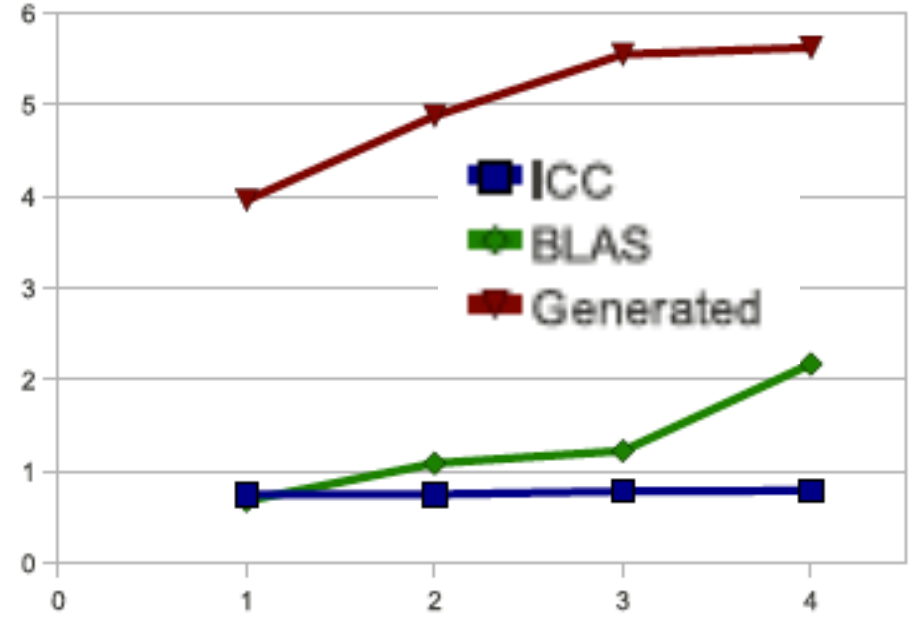
```
1: procedure IKJ(A_ki, B_jk, C_ij)
2:
3:     for (i ← 0 ; i < M ; i++) do
4:         for (k ← 0 ; k < K ; k+= 4) do
5:             a_0[0:3] ← SPLAT(A[k+0][i])
6:             a_1[0:3] ← SPLAT(A[k+1][i])
7:             a_2[0:3] ← SPLAT(A[k+2][i])
8:             a_3[0:3] ← SPLAT(A[k+3][i])
9:             for (j ← 0 ; j < N ; j+= 4) do
10:                b_0[0:3] ← B[j+0][k:k+3]
11:                b_1[0:3] ← B[j+1][k:k+3]
12:                b_2[0:3] ← B[j+2][k:k+3]
13:                b_3[0:3] ← B[j+3][k:k+3]
14:                TRANSPOSE(b_0, b_1, b_2, b_3)
15:                c[0:3] ← C[i][j:j+3]
16:                c[0:3]+= a_0[0:3] * b_0[0:3]
17:                c[0:3]+= a_1[0:3] * b_1[0:3]
18:                c[0:3]+= a_2[0:3] * b_2[0:3]
19:                c[0:3]+= a_3[0:3] * b_3[0:3]
20:                C[i][j:j+3] ← c[0:3]
21:            end for
22:        end for
23:    end for
24: end procedure
```

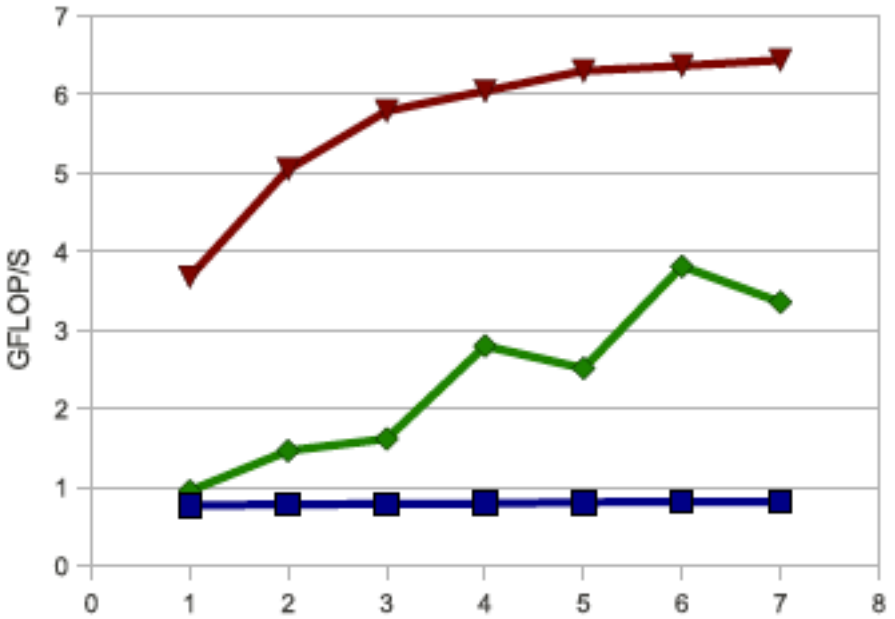
Contraction

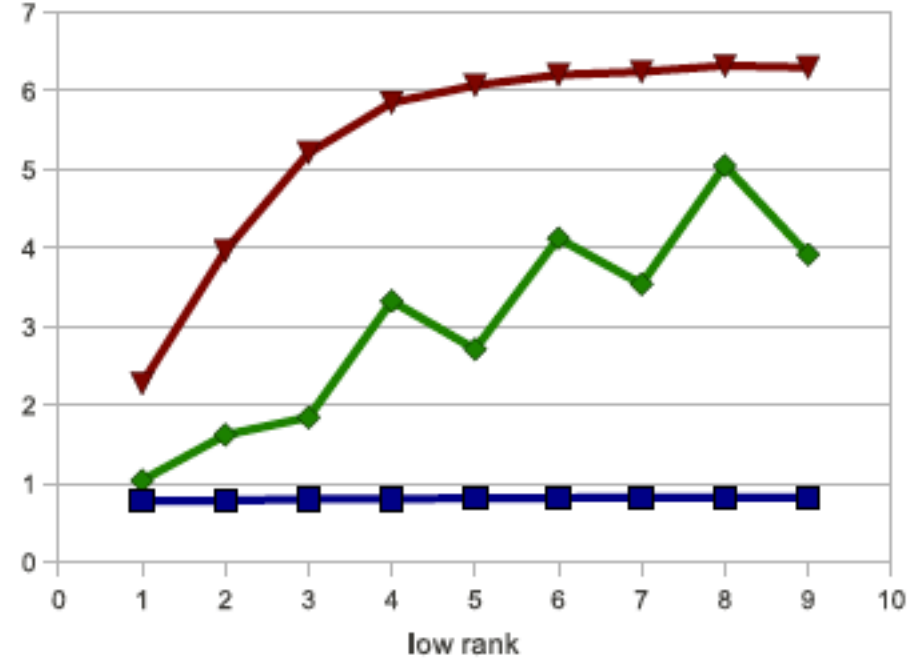
$$C_{ij} = \sum_k A_{ki} \cdot B_{jk}$$

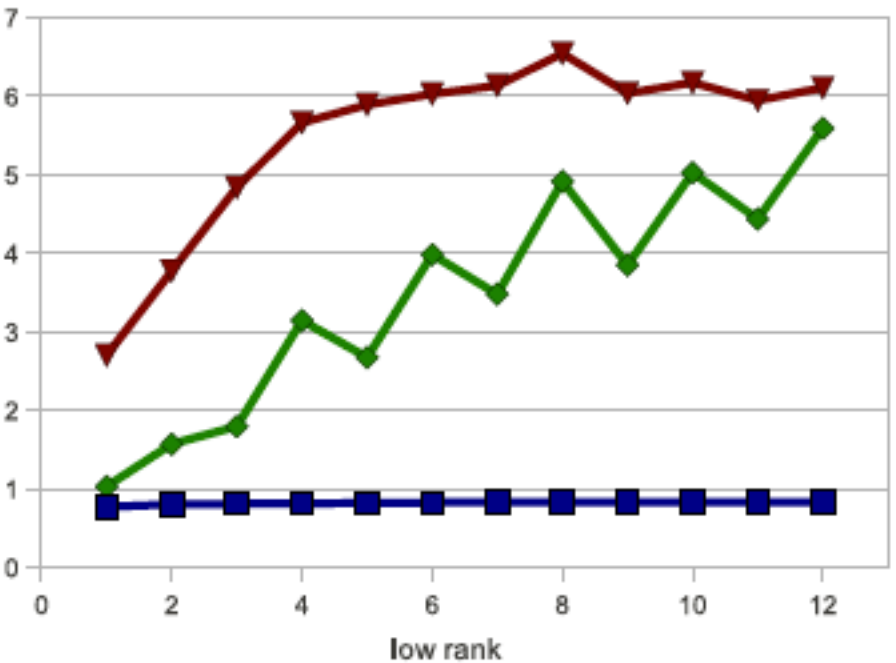
- Vectorized along j
- B_{jk} transposed
- Each element of A_{ki} is splatted (broadcast) to all elements of a vector register

Multiresolution Kernel Performance









Performance: CCSD Tensor Contraction

$$C_{ijkl} = \sum_{mn} A_{imkn} \cdot B_{jnlm}$$

```
for (i=0; i<P; i++)
 for (j=0; j<Q; j++)
  for (k=0; k<R; k++)
   for (l=0; l<S; l++)
    for (m=0; m<T; m++)
     for (n=0; n<U; n++)
      C[i][j][k][l] += A[i][m][k][n]*B[j][n][l][m];
```

Configuration	GCC	ICC	Machine Peak	Optimized
Nehalem double sse	1.406	2.740	10.64	7.579
Nehalem single sse	1.405	2.642	21.28	13.428
Sandy Bridge double avx	2.231	4.361	27.2	16.768
Sandy Bridge single avx	2.255	5.075	54.4	36.937

Performance in GFLOP/s

Domain-Specific Optimization: Stencils

Carnegie Mellon University
Franz Franchetti,
Richard Veras

Ohio State University
Tom Henretty, Justin Holewinski, Martin Kong,
Nasko Rountev,
P. Sadayappan

Louisiana State University
J. Ramanujam

UCLA
Louis-Noel Pouchet

Supported by NSF and DOE

Embedded DSL for Stencils

- Benefits of high-level specification of computations
  - Ease of use
    - For mathematicians/scientists creating the code
  - Ease of optimization
    - Facilitate loop and data transformations by compiler
    - Automatic transformation by compiler into parallel C/C++ code
- Embedded DSL provides flexibility
  - Generality of standard programming language (C, MATLAB) for non compute-intensive parts
  - Automated transformation of embedded DSL code for high performance on different target architectures
- Target architectures for Stencil DSL
  - Vector-SIMD (AVX, LRBNi, ..), GPU, FPGA, customized accelerators

Stencil DSL Example -- Standalone

```
int Nr; int Nc;
grid g [Nr][Nc];

double griddata a on g at 0,1;

pointfunction five_point_avg(p) {
  double ONE_FIFTH = 0.2;
  [1]p[0][0] = ONE_FIFTH*([0]p[-1][0] + [0]p[0][-1]
                 + [0]p[0][0] + [0]p[0][1] + [0]p[1][0]);
}

iterate 1000 {
  stencil jacobi_2d {
    [0     ][0:Nc-1] : [1]a[0][0] = [0]a[0][0];
    [Nr-1  ][0:Nc-1] : [1]a[0][0] = [0]a[0][0];
    [0:Nr-1][0     ] : [1]a[0][0] = [0]a[0][0];
    [0:Nr-1][Nc-1  ] : [1]a[0][0] = [0]a[0][0];
    [1:Nr-2][1:Nc-2] : five_point_avg(a);
  }

  reduction max_diff max {
    [0:Nr-1][0:Nc-1] : fabs([1]a[0][0] - [0]a[0][0]);
  }
} check (max_diff < .00001) every 4 iterations
```

Reference data over two time steps: current(0) and next (1)

Stencil DSL – Embedded in C

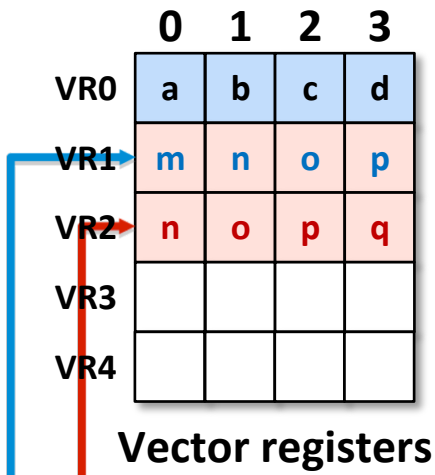
```
int main() {
  int Nr = 256; int Nc = 256; int T = 100;
  double *a = malloc(Nc*Nr*sizeof(double));

#pragma sdsl start time_steps:T block:8,8,8 tile:1,3,1 time:4
  int Nr; int Nc;
  grid g [Nr][Nc];
  double griddata a on g at 0,1;
  pointfunction five_point_avg(p) {
    double ONE_FIFTH = 0.2;
    [1]p[0][0] = ONE_FIFTH*([0]p[-1][0] + [0]p[0][-1]
                + [0]p[0][0] + [0]p[0][1] + [0]p[1][0]); }
  iterate 1000 {
    stencil jacobi_2d {
      [0     ][0:Nc-1] : [1]a[0][0] = [0]a[0][0];
      [Nr-1  ][0:Nc-1] : [1]a[0][0] = [0]a[0][0];
      [0:Nr-1][0     ] : [1]a[0][0] = [0]a[0][0];
      [0:Nr-1][Nc-1  ] : [1]a[0][0] = [0]a[0][0];
      [1:Nr-2][1:Nc-2] : five_point_avg(a);}
    reduction max_diff max {
      [0:Nr-1][0:Nr-1] : fabs([1]a[0][0] - [0]a[0][0]);
    }
  } check (max_diff < .00001) every 4 iterations
  #pragma sdsl end
}
```

Related Work

- 20+ publications over the last few years on optimizing stencil computations
- Some stencil DSLs and stencil compilers
  - Pochoir (MIT), PATUS (Univ. Basel), Halide (MIT)
- Frameworks for building DSLs
  - SEJITS (LBL); Liszt, OptiML, OptiQL, ... (Stanford)
- Our focus has been complementary: developing abstraction-specific compiler transformations matched to performance-critical characteristics of target architecture

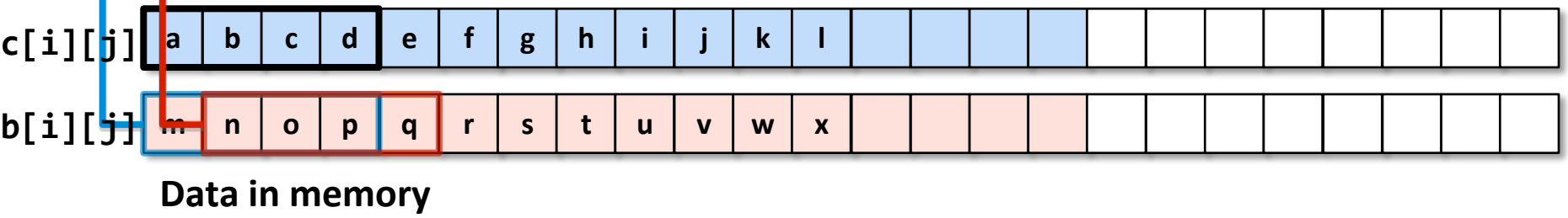
Stencils on Vector-SIMD Processors

```
for (i=0; i<H; ++i)
 for (j=0; j<W; ++j)
  c[i][j]+=b[i][j]+b[i][j+1];
```



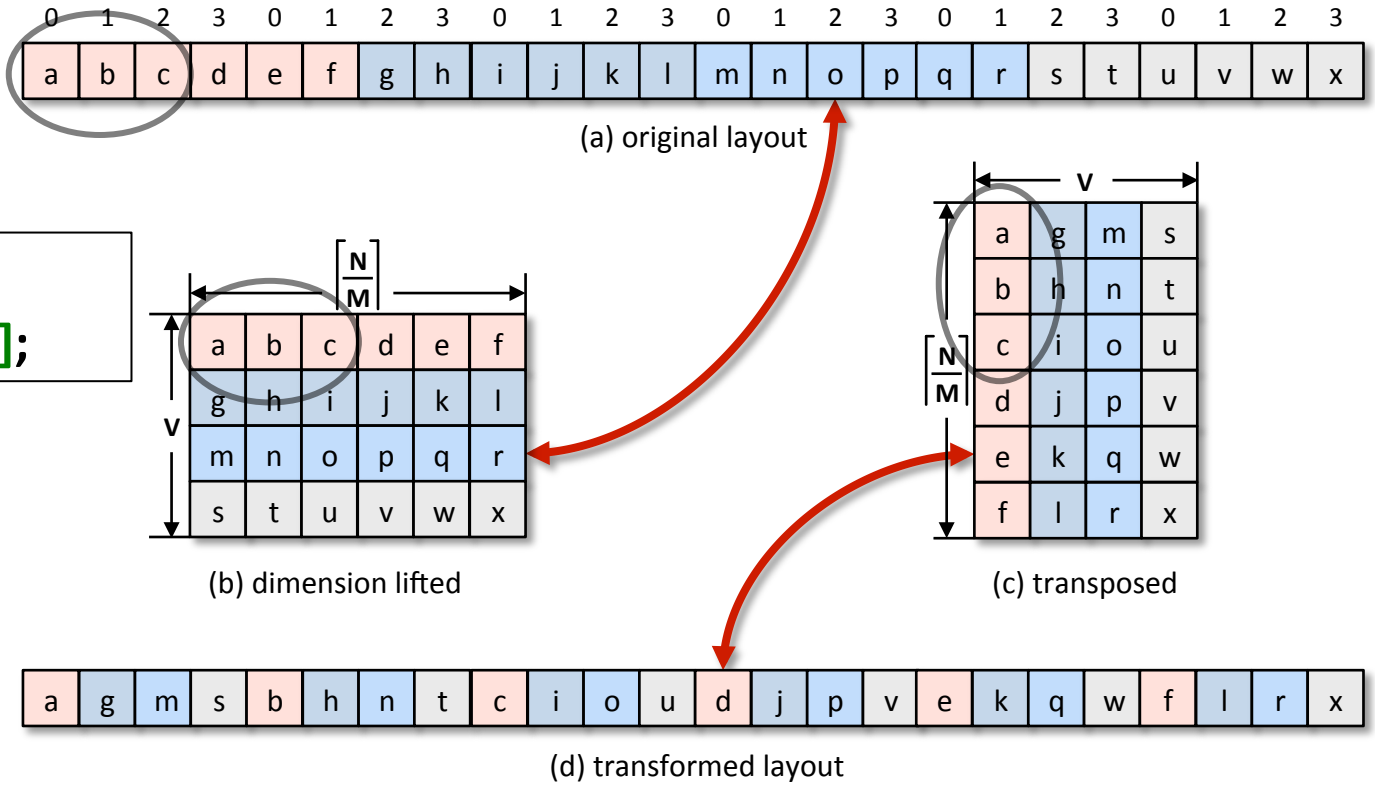
- Fundamental source of inefficiency with stencil codes on current short-vector SIMD ISAs (e.g. SSE, AVX …)
  - Concurrent operations on contiguous elements
  - Each data element is reused in different "slots" of vector register
  - Redundant loads or shuffle ops needed
- Compiler transformations based on matching computational characteristics of stencils to vector-SIMD architecture characteristics

Inefficiency: Each element of b is loaded twice



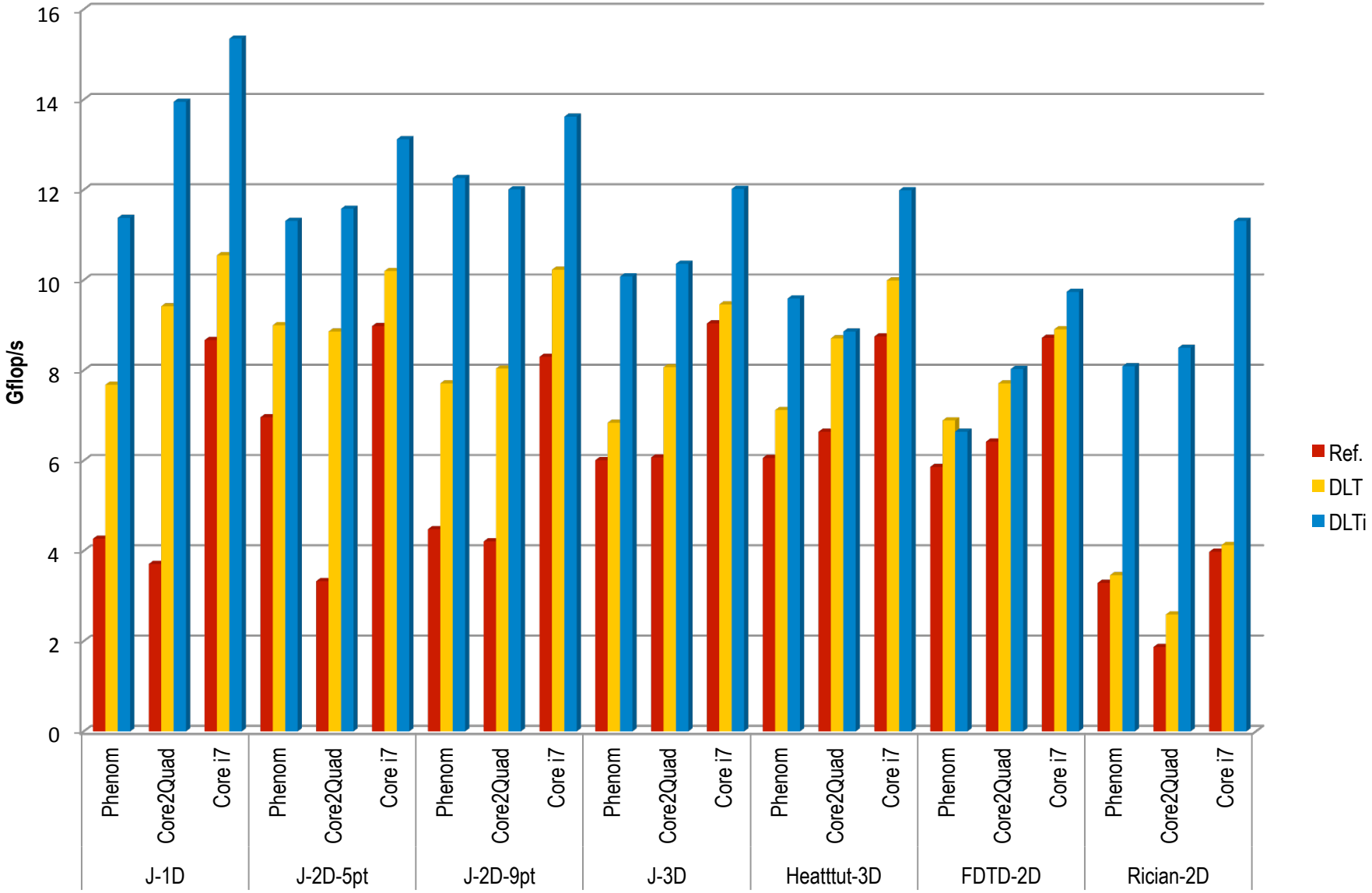
Data Layout Transformation

```
for (i = 0; i < N; ++i)
  a[i]=b[i-1]+b[i]+b[i+1];
```

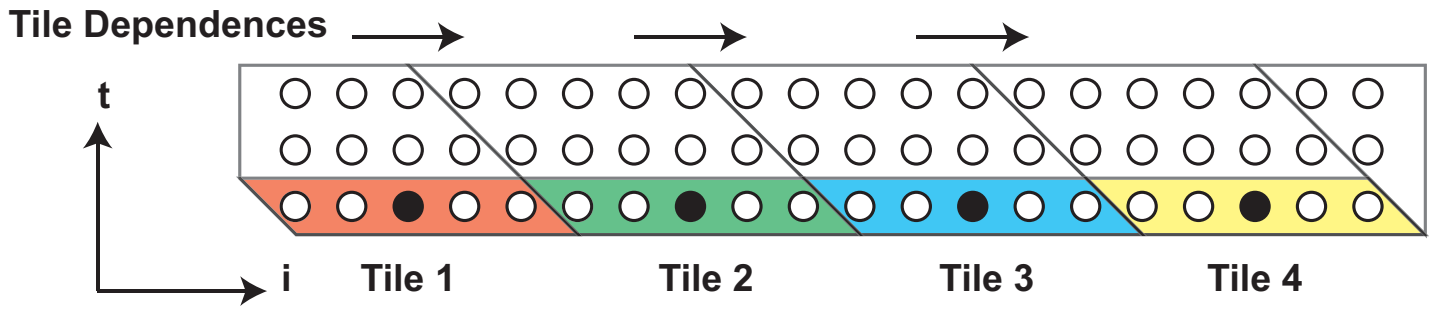


- 1D vector in memory ⇔ (b) 2D logical view of same data
- (c) Transposed 2D array moves interacting elements into same slot of different vectors ⇔ (d) New 1D layout after transformation
- Boundaries need special handling

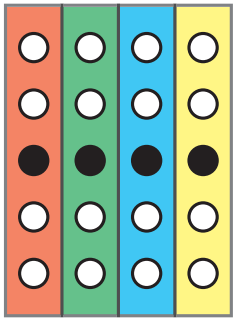
Data Layout Transformation: Evaluation



Standard Tiling with DLT



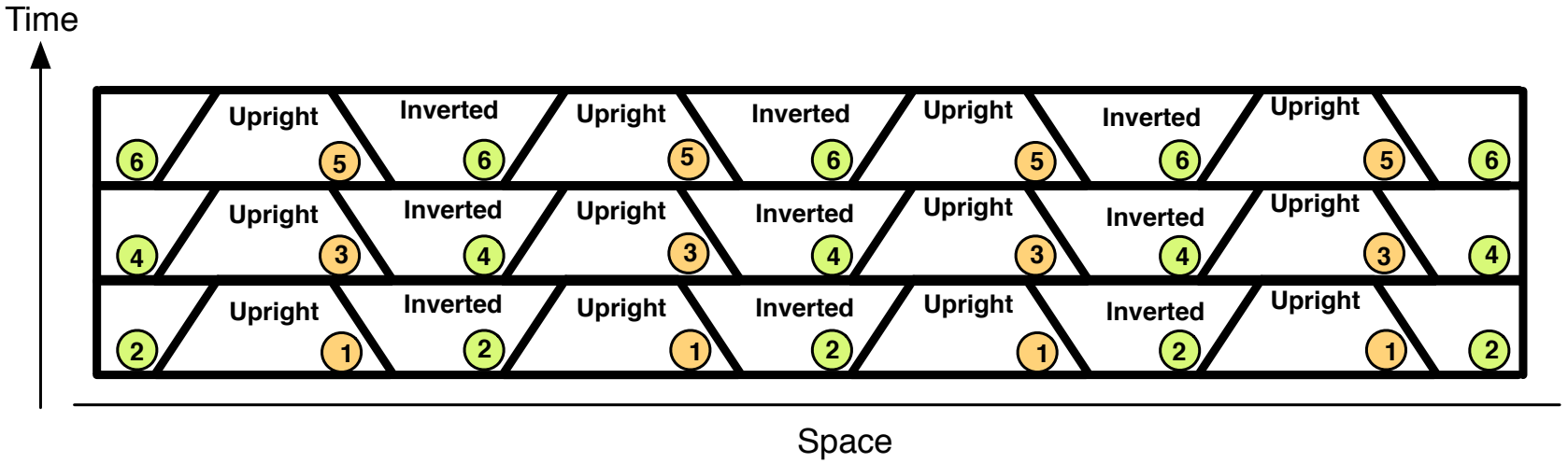
(a) Standard tiling -- Linear view



(b) Standard tiling -- DLT view (t=1)

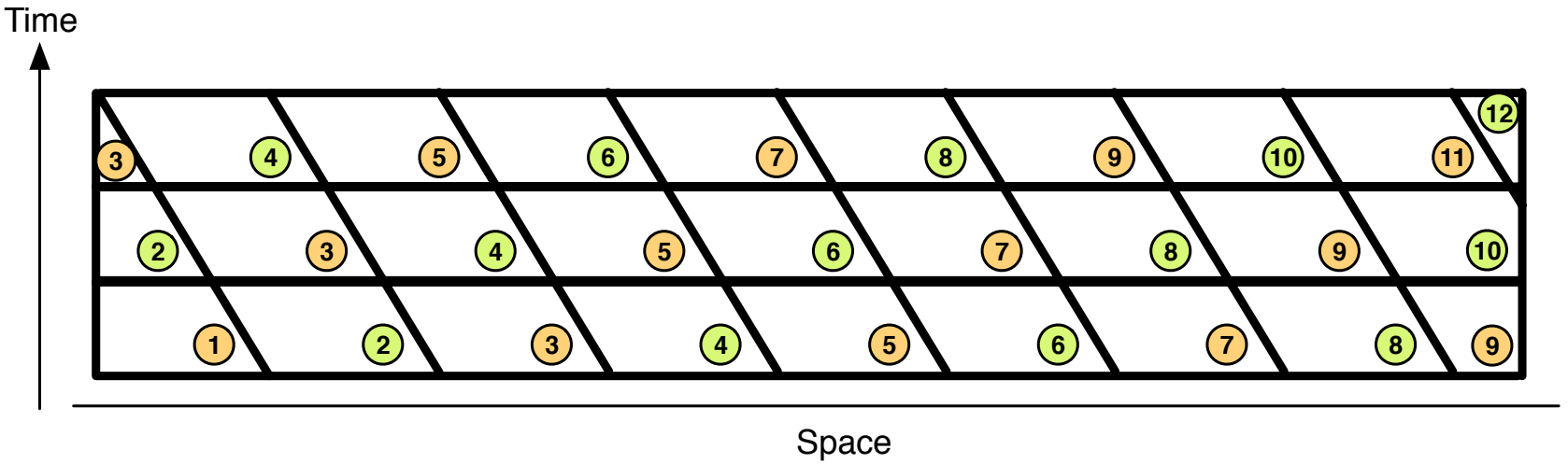
- Standard tiling cannot be used with the layout transform
- Inter-tile dependences prevent vectorization

Split-tiling



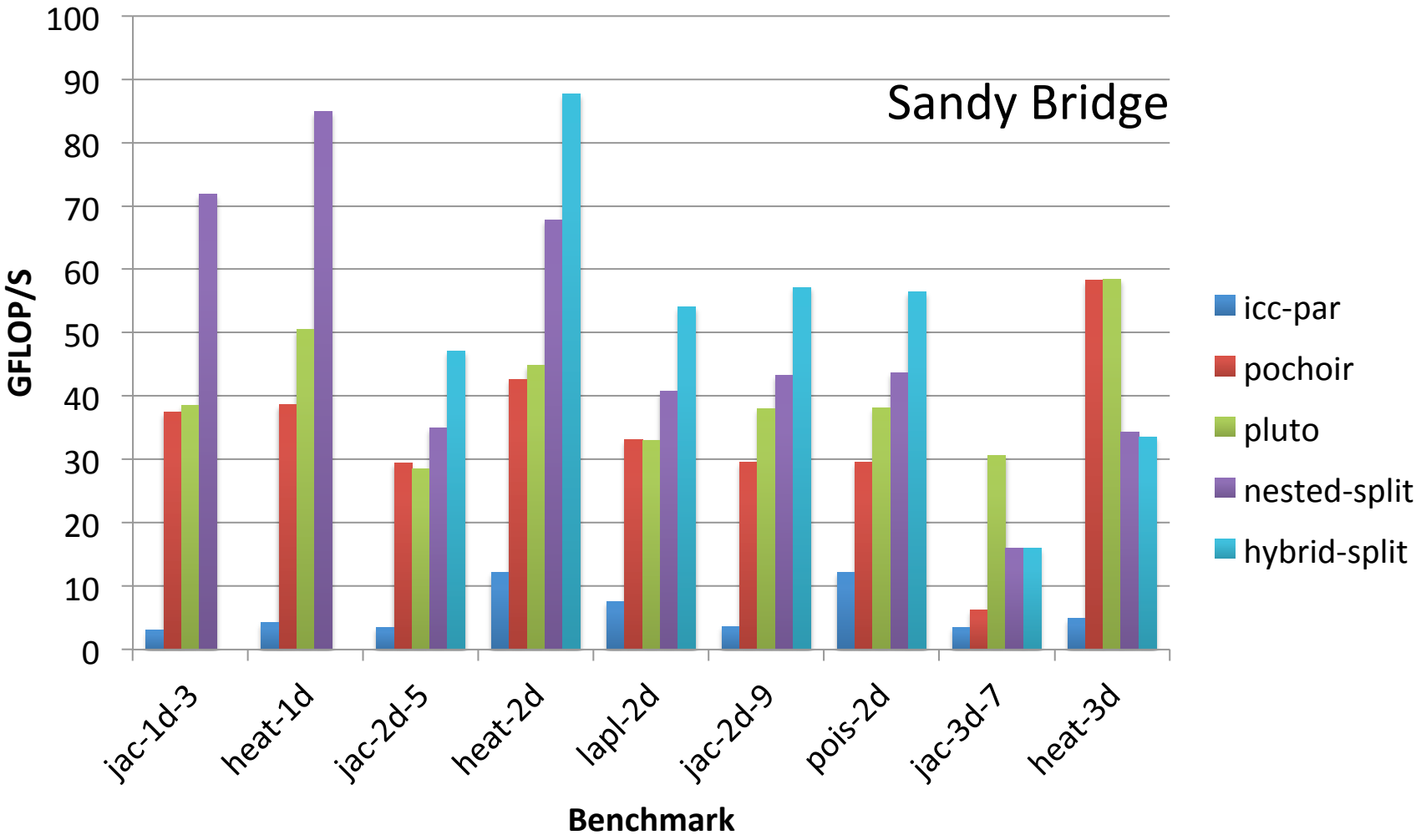
- Divide iteration space into upright and inverted tiles
- For each *tt* timesteps where *tt* = time tile size…
  - Execute all upright (U) tiles in parallel
  - Execute all inverted (I) tiles in parallel
- Nested split tiling can be used for multiple spatial dimensions
  - For 2D, 4 kinds of tiles: UU, UI, IU, II

Hybrid Split-tiling



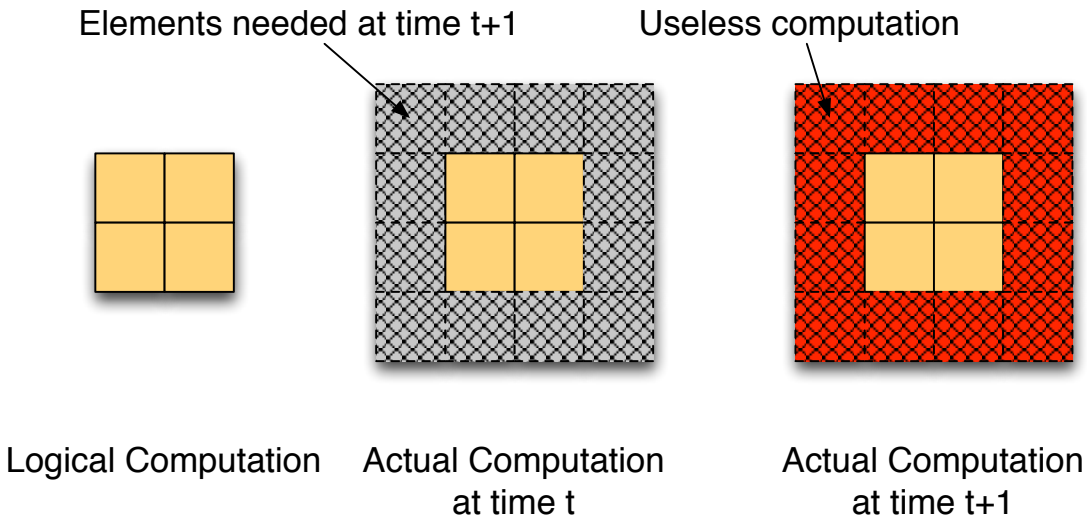
- Trade-off between standard tiling (parallelogram shape) and split tiling (trapezoidal shape)
  - Split tiling enables inter-tile parallelism and DLT but cache footprint grows with time-tile size
  - Standard tiling inhibits inter-tile parallelism but cache footprint is independent of time-tile size
- Hybrid approach: combine split tiling along some inner spatial loops and standard tiling along outer spatial loops

Experimental Results: DLT+Split Tiling

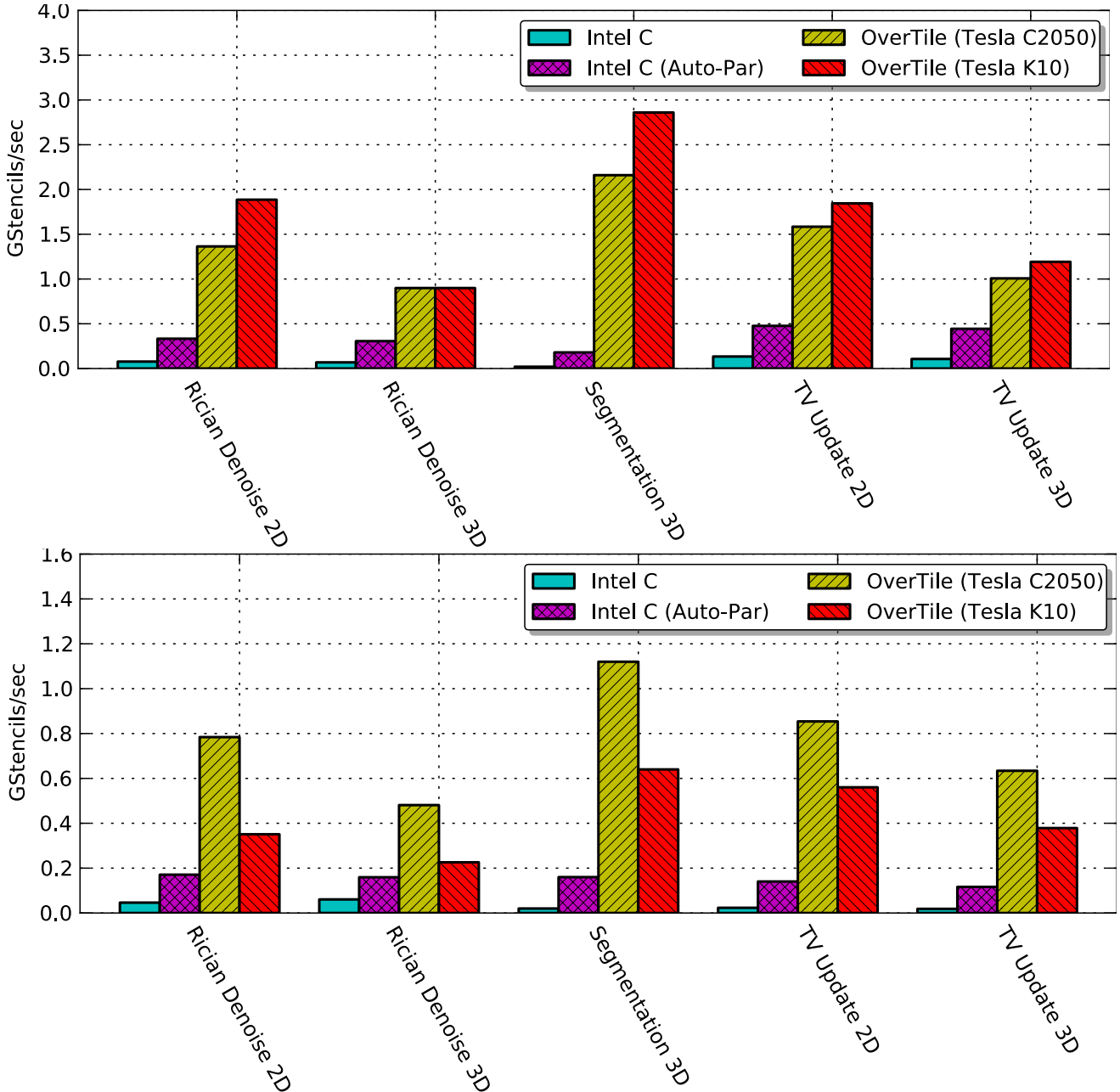


Stencils on GPUs

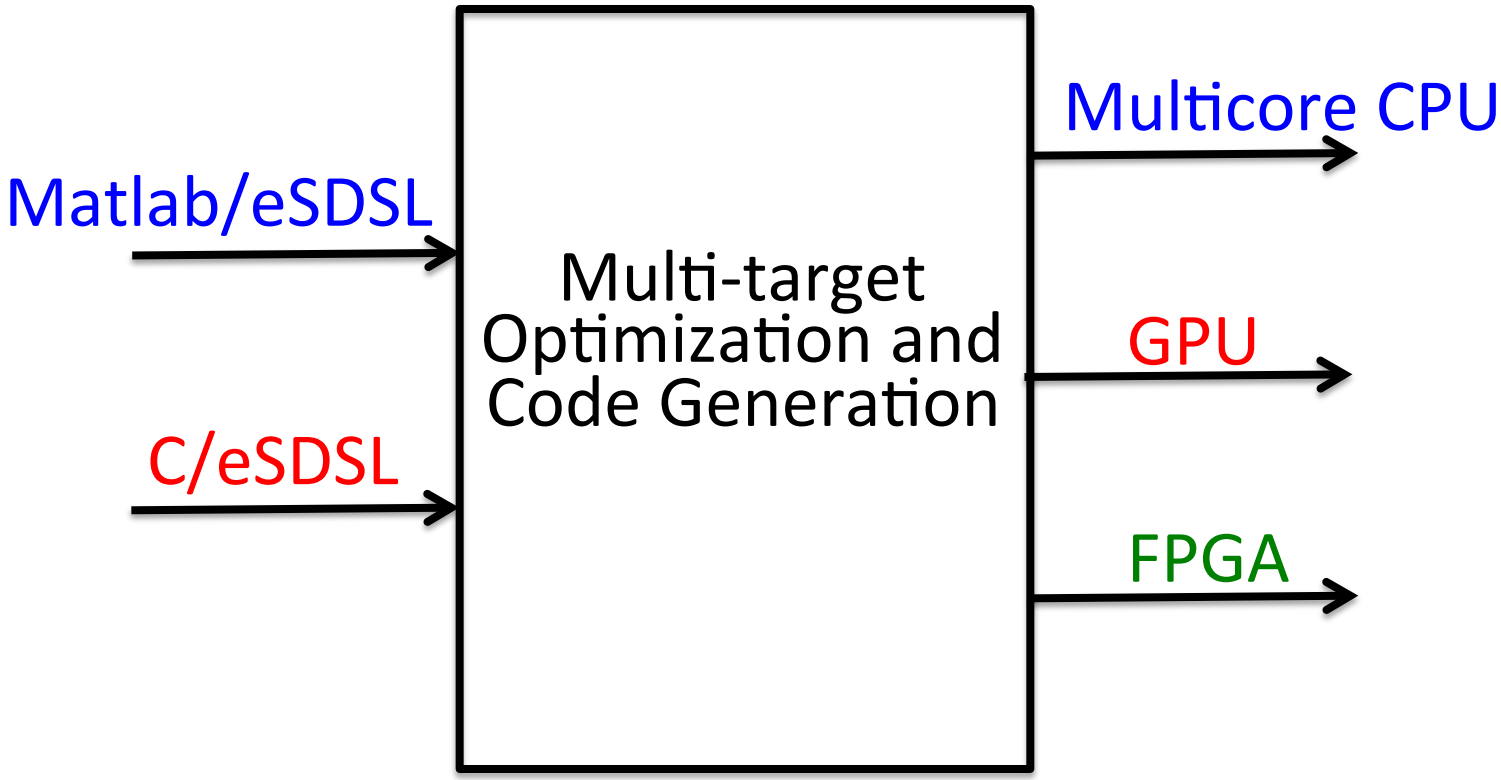
- Vector-SIMD alignment problems non-existent
- Different optimization challenges: limited forms of synchronization, avoidance of thread divergence
- Overlapped tiling
  - Redundantly compute neighboring cells to avoid inter-thread-block sync, lower communication, and avoid thread divergence



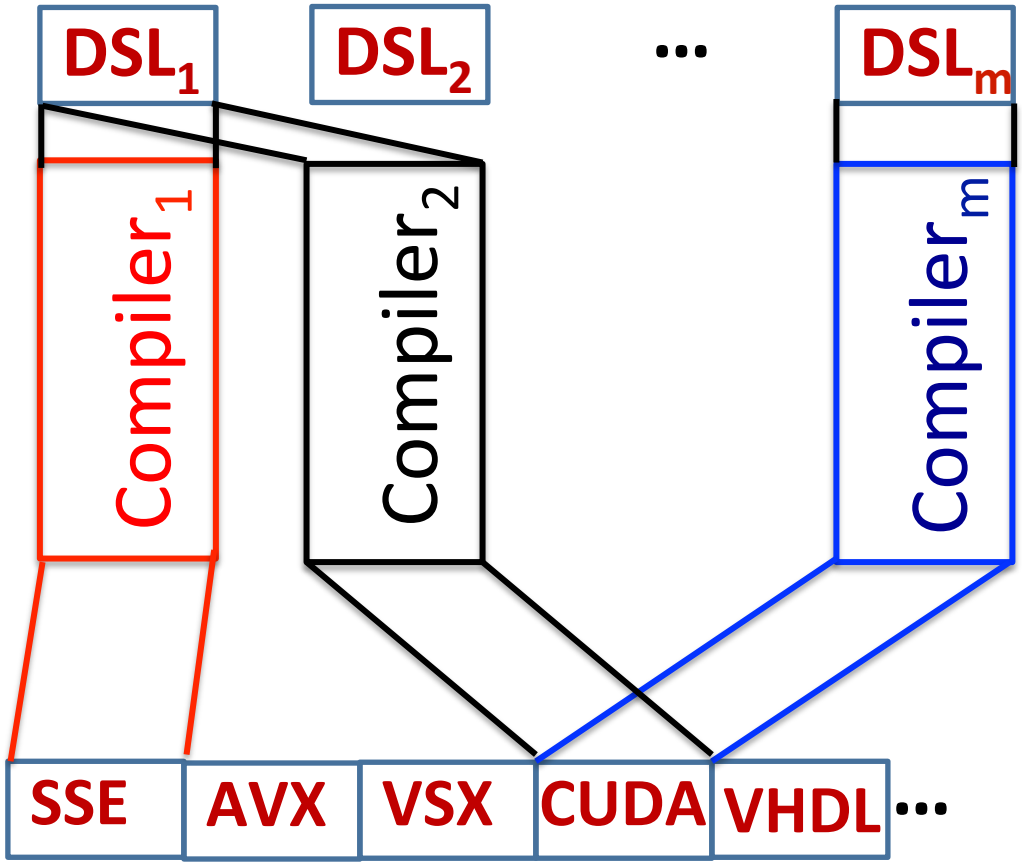
Stencil Compiler for GPU: Performance



Multi-target Code Generation from SDSL

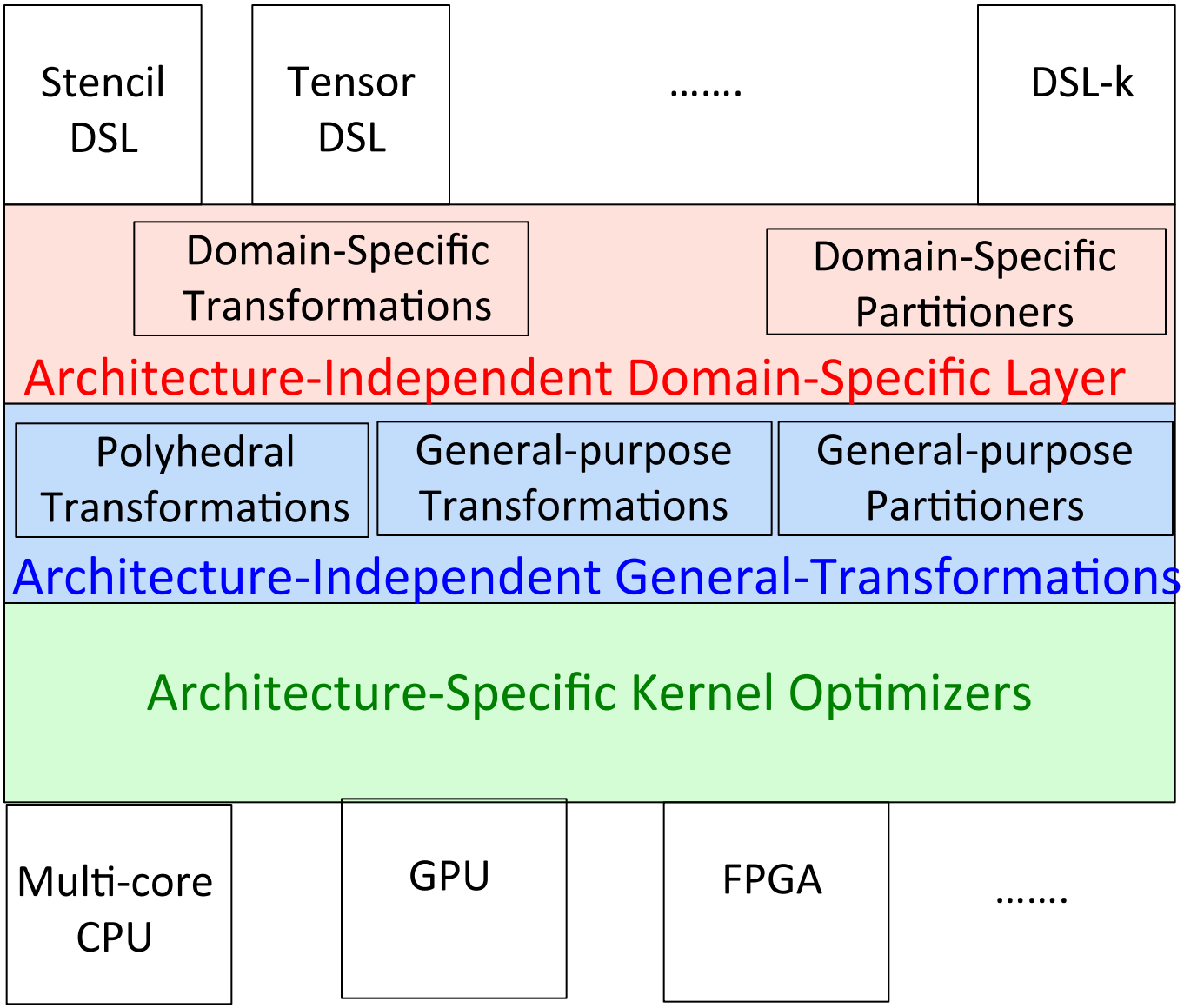


Many Stand-Alone DSL Efforts



- Our DSL compilers for tensors and stencils did not reuse components
- Brute-force approach: N*M compilers for N domains and M platforms
- Can we achieve reuse in optimizing many DSLs for multiple targets?

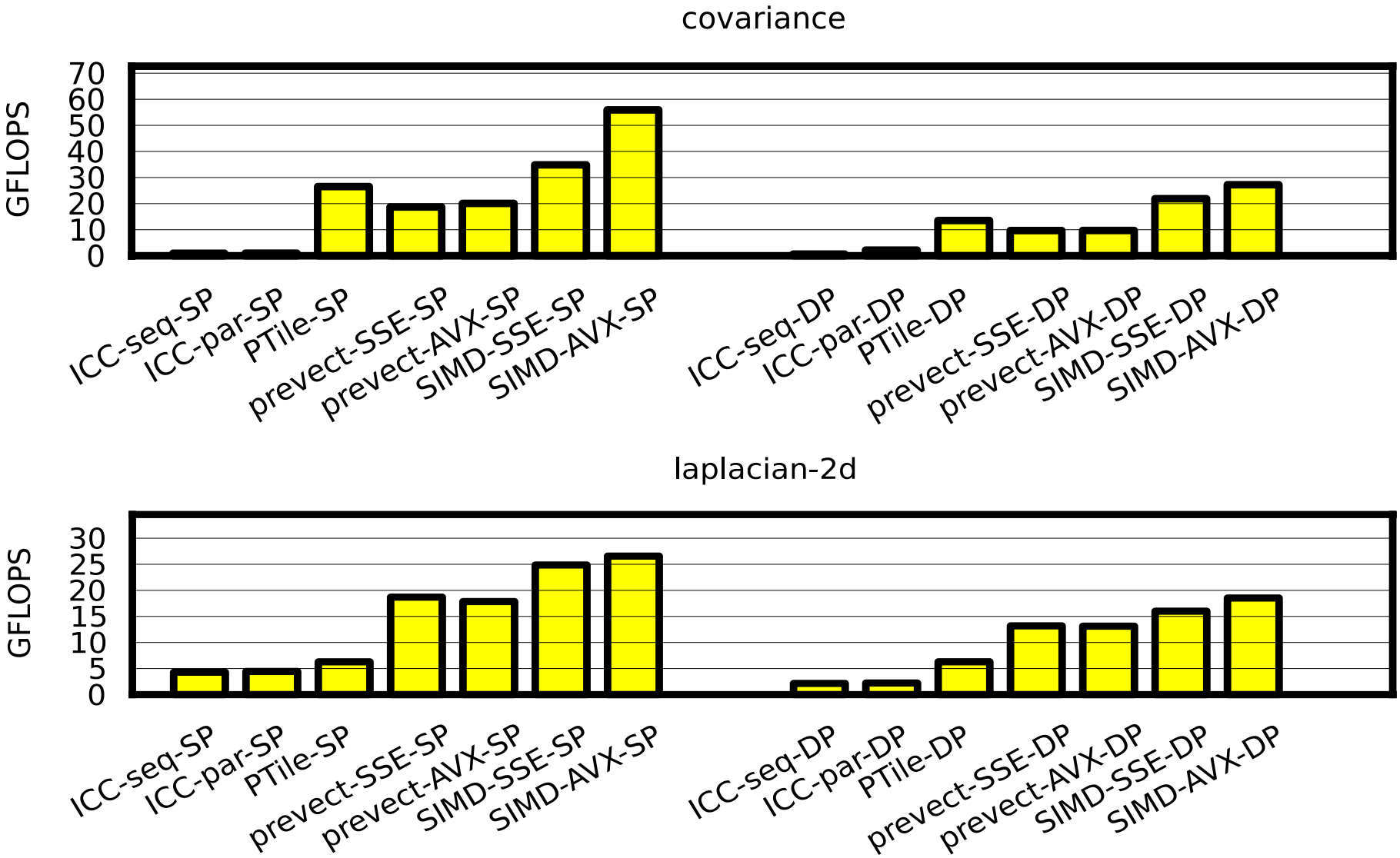
A Layered Approach



Codelet-Centric Vector-SIMD Optimization

- Problem: Compilers like icc typically achieve only a fraction of processor's peak performance on polyhedrally transformed code
  - Overheads due to unaligned load/store operations
  - Overheads due to repeated load/stores of reused data elements
- Solution: Develop a decoupled two-step optimization process that allows separation of concerns:
  - Use polyhedral compiler transformations to form small (L1 cache resident) tiles with specific desirable properties: vector codelet
  - Use a low-level code optimizer (SPIRAL back-end) to generate optimized code for vector codelet
- Properties of vectorizable codelet: formulated as an Int. Lin. Prog.
  - Maximal inner-most loop parallelism
  - Maximal number of stride 0/1 accesses in inner-most loop
  - Maximize number of innermost permutable loops
  - Minimization of unaligned load/store operations
- Same approach now being targeted customized accelerators
- Details in PLDI '13 paper

Experimental Results: Vector Codelets



IO Complexity: Revisiting the Red/Blue Pebble Game

ENS
Fabrice Rastello

Ohio State University
Venmugil Elango,
P. Sadayappan

Louisiana State University
J. Ramanujam

UCLA
Louis-Noel Pouchet

High-Level Summary

- Characterizing "communication" complexity is much more complicated than computational complexity
  - Number of data transfers from large "slow" memory to limited "fast" memory (cache, registers etc.)
  - Unlike comp. complexity, depends on order of execution of operations and amount of fast memory
- First IO lower bounds: Hong/Kung classic 1981 paper
  - Tight lower bounds for some algorithms but loose for others
  - Monolithic: Cannot compose analyses: S = S1;S2
  - Only used for analysis of regular Computational DAGs
  - Recent advance by Demmel, Yelick, and colleagues at Berkeley provides a generalization for a class of loop computations (linear and tensor algebra, N-body, etc.), but not effective for stencils etc
- We develop pebbling variant and new bounding technique
  1. Enables composability: LB(S) from LB(S1) and LB(S2), S = S1;S2
  2. Tighter lower bounds for complementary class of CDAGs
  3. Enables full automation for arbitrary, irregular CDAGs

CDAG characterization [Bilardi 2001]

A computation directed acyclic graph (CDAG) is a 4-tuple $C = (I, V, E, O)$ of finite sets such that:

1. $I \cap V = \emptyset$;
2. $E \subseteq (I \cup V) \times V$ is the set of arcs;
3. $G = (I \cup V, E)$ is a directed acyclic graph with no isolated vertices;
4. I is called the input set;
5. V is called the operation set and all its vertices have one or two incoming arcs;
6. $O \subseteq (I \cup V)$ is called the output set.

Hong-Kung (1981): Red-Blue Pebble Game

Given a CDAG G, S red pebbles and an arbitrary number of blue pebbles, with a blue pebble on each *input* vertex, a complete game is any sequence of steps obeying the following rules that results a final state with blue pebbles on all *output* vertices:

R1 (Input) A red pebble may be placed on any vertex that has a blue pebble (load from slow to fast memory),

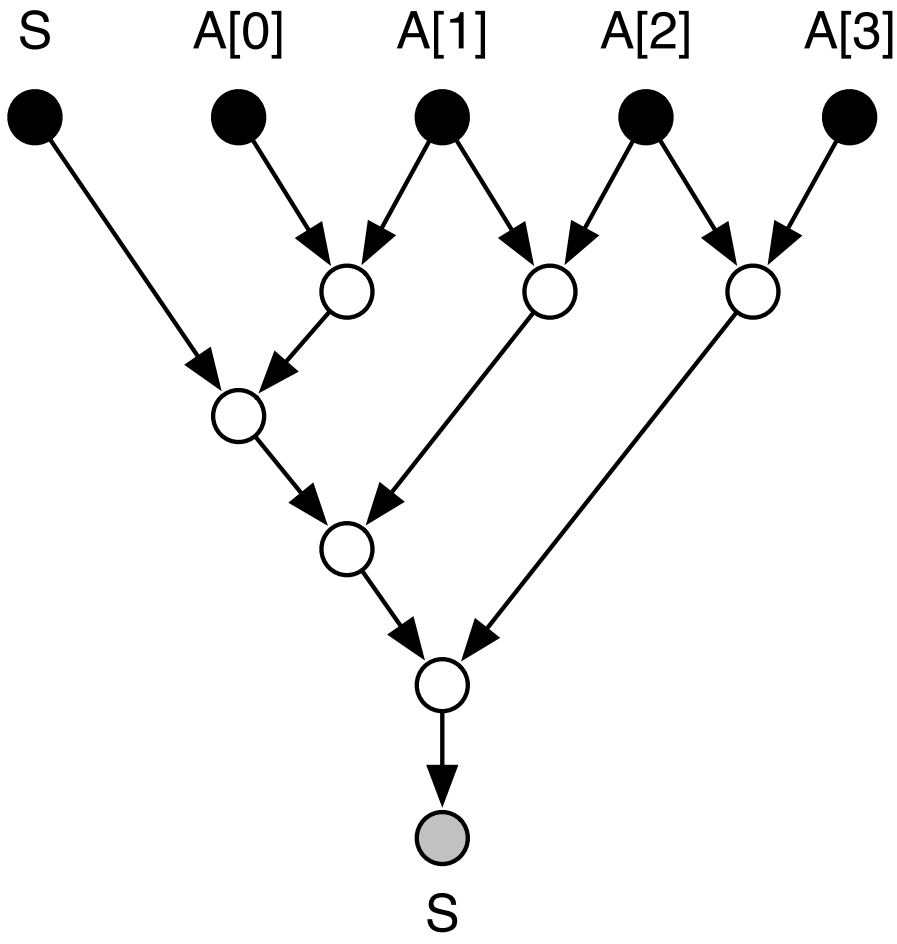
R2 (Output) A blue pebble may be placed on any vertex that has a red pebble (store from fast to slow memory),

R3 (Compute) If all immediate predecessors of a vertex have red pebbles, a red pebble may be placed on that vertex (execution or "firing" of operation),

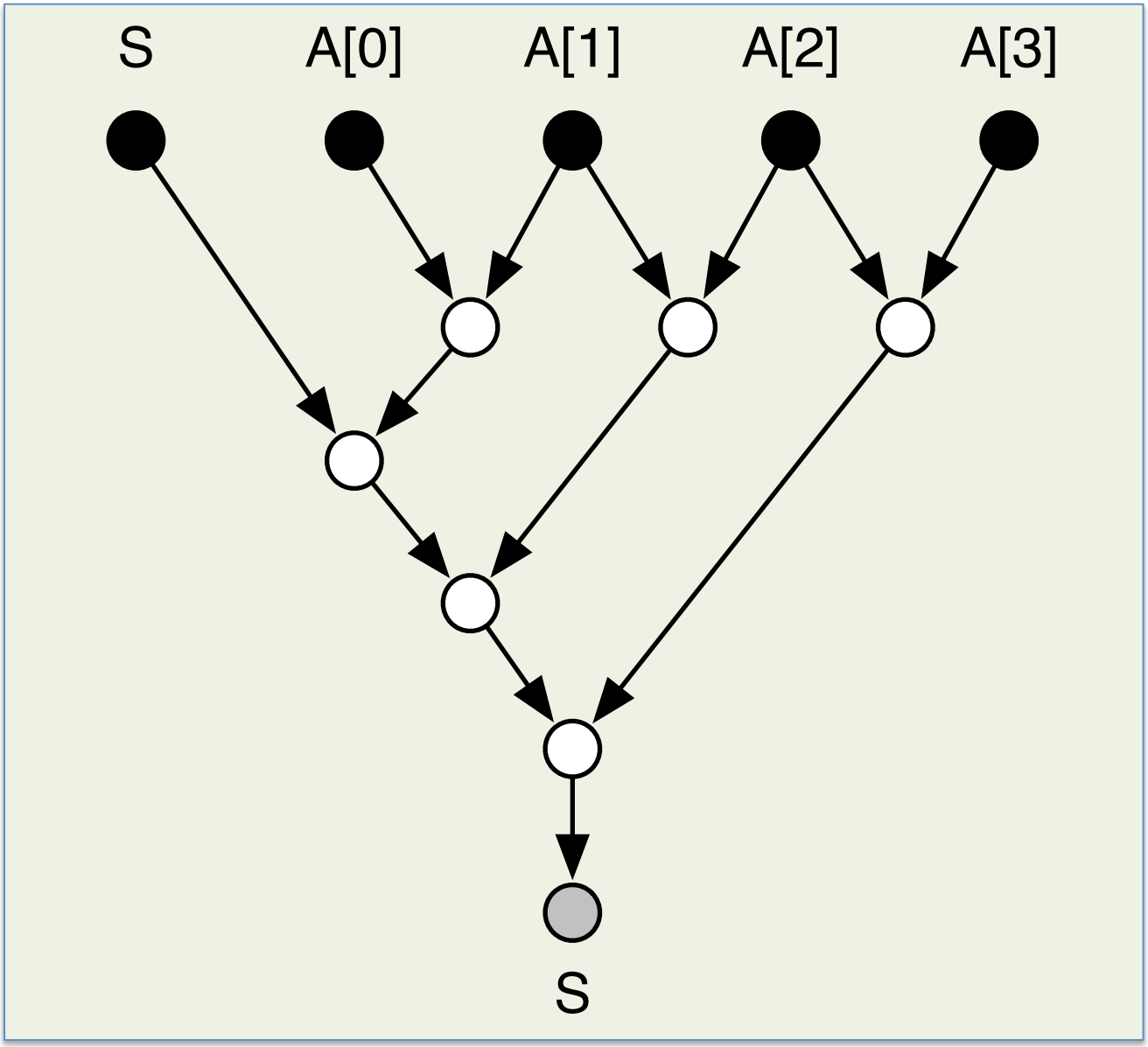
R4 (Delete) A red pebble may be removed from any vertex (reuse storage).

Code Example and its CDAG

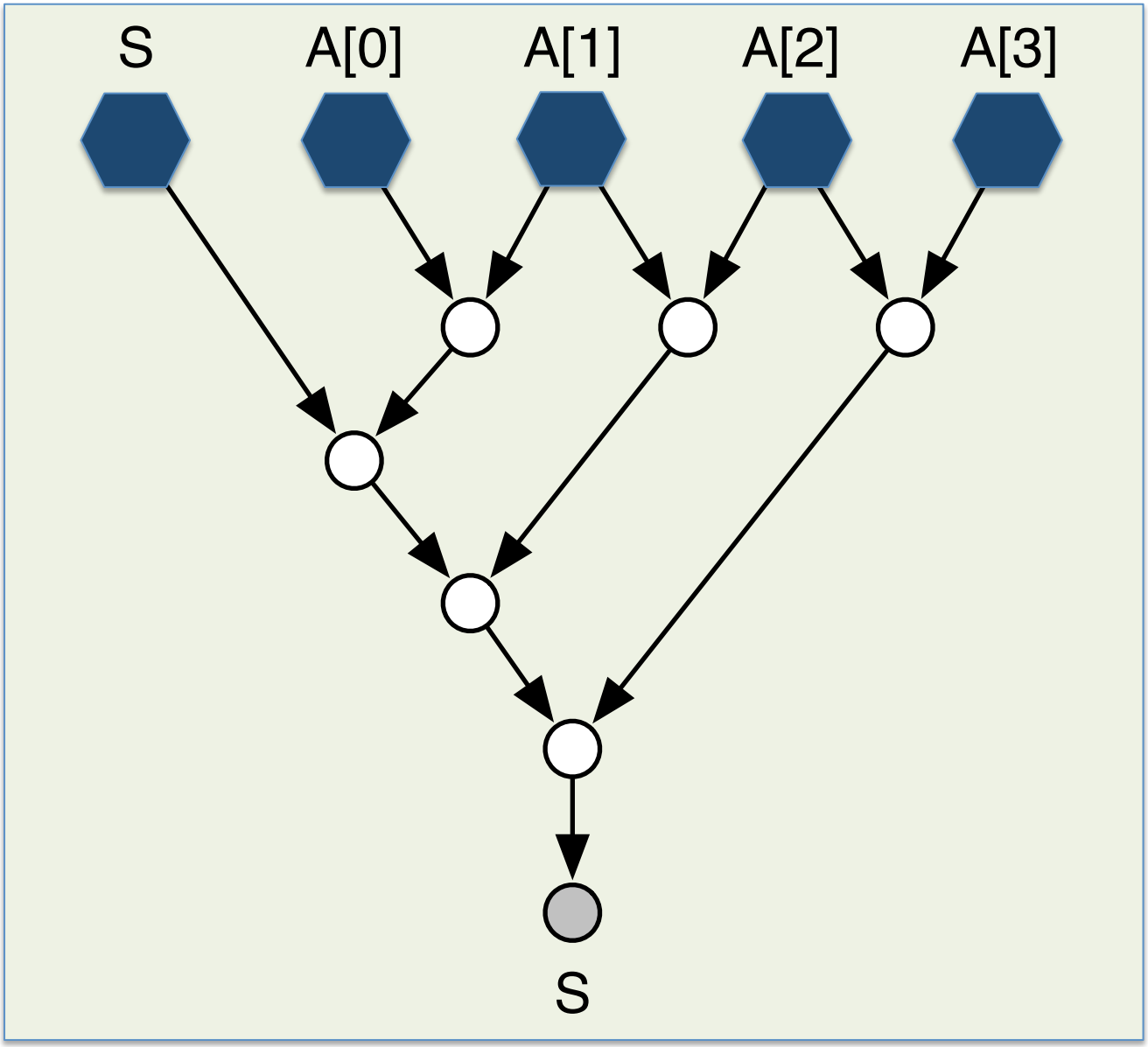
```
for (i = 1; i < 4; ++i)
  S += A[i-1] * A[i];
```



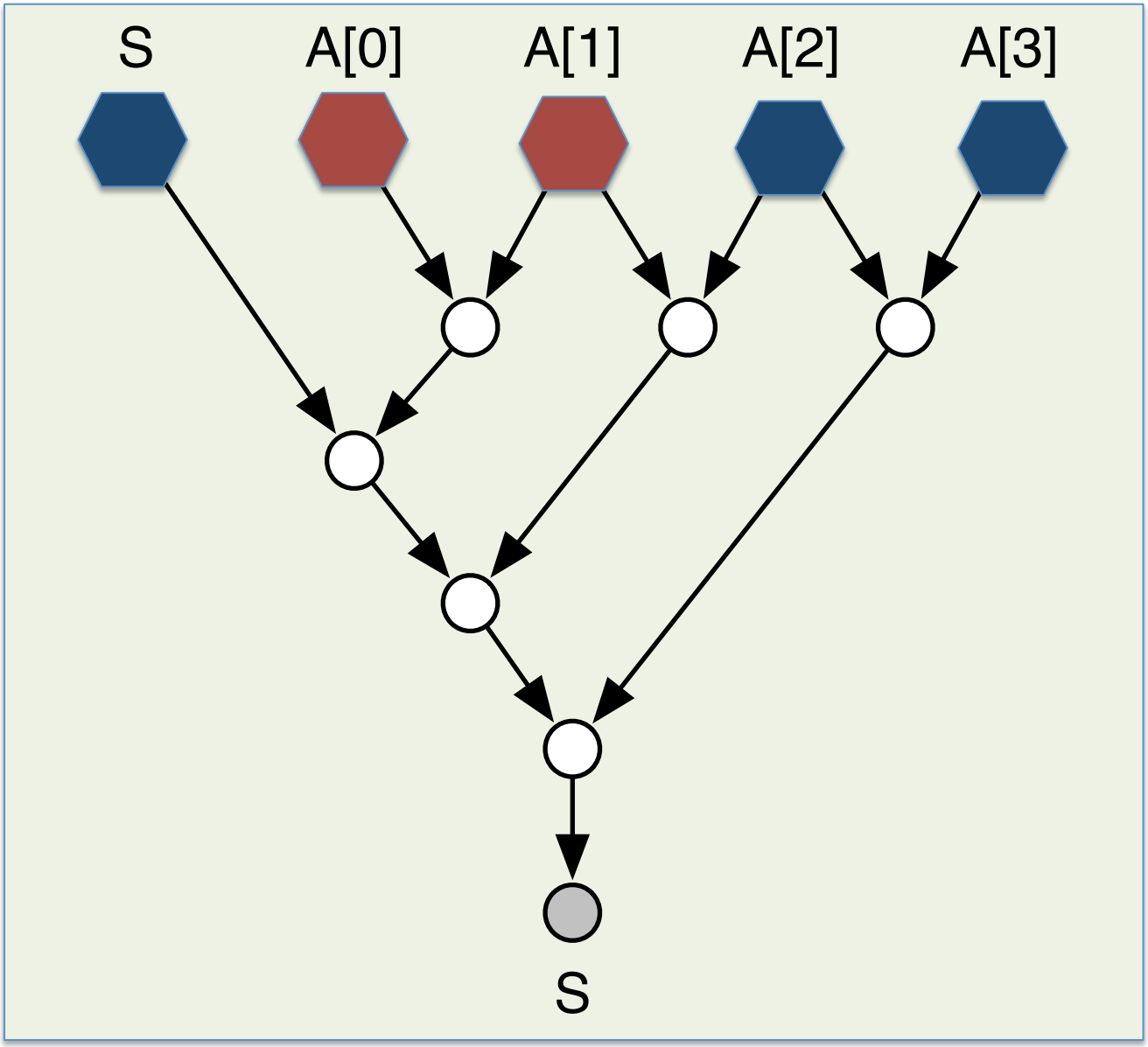
Red-Blue Pebble Game: 2 Red Pebbles



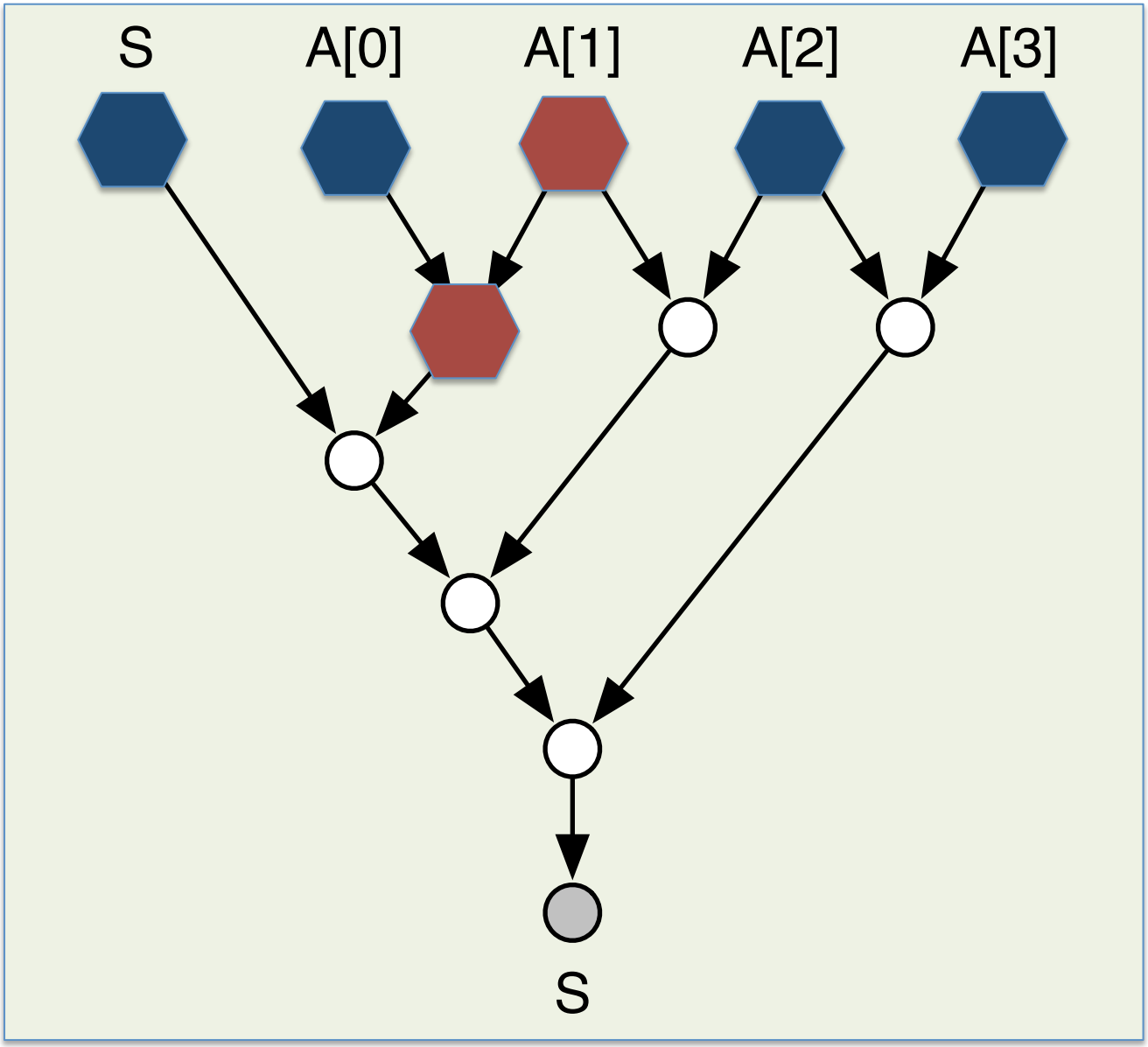
Red-Blue Pebble Game: 2 Red Pebbles



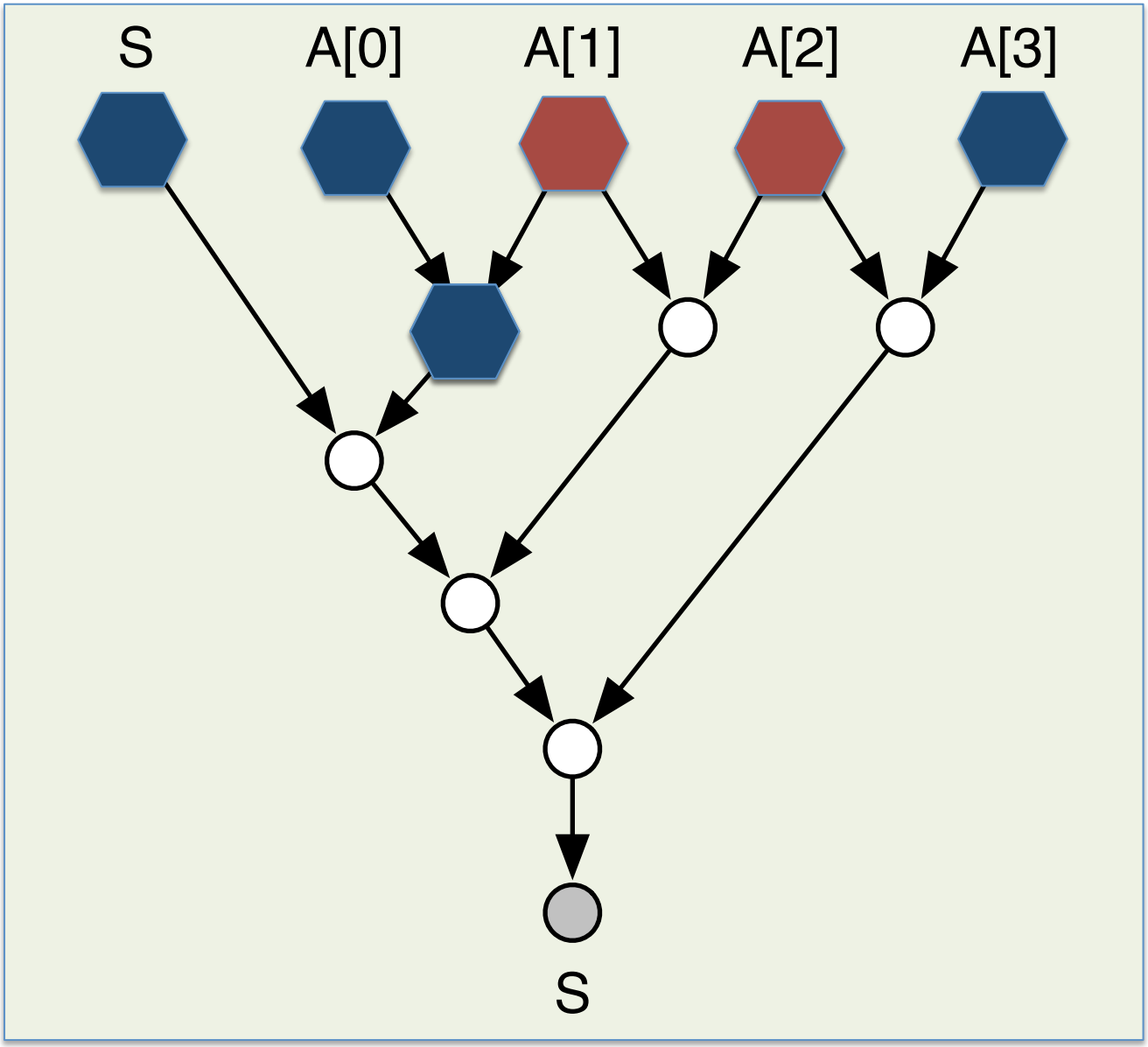
Red-Blue Pebble Game: 2 Red Pebbles



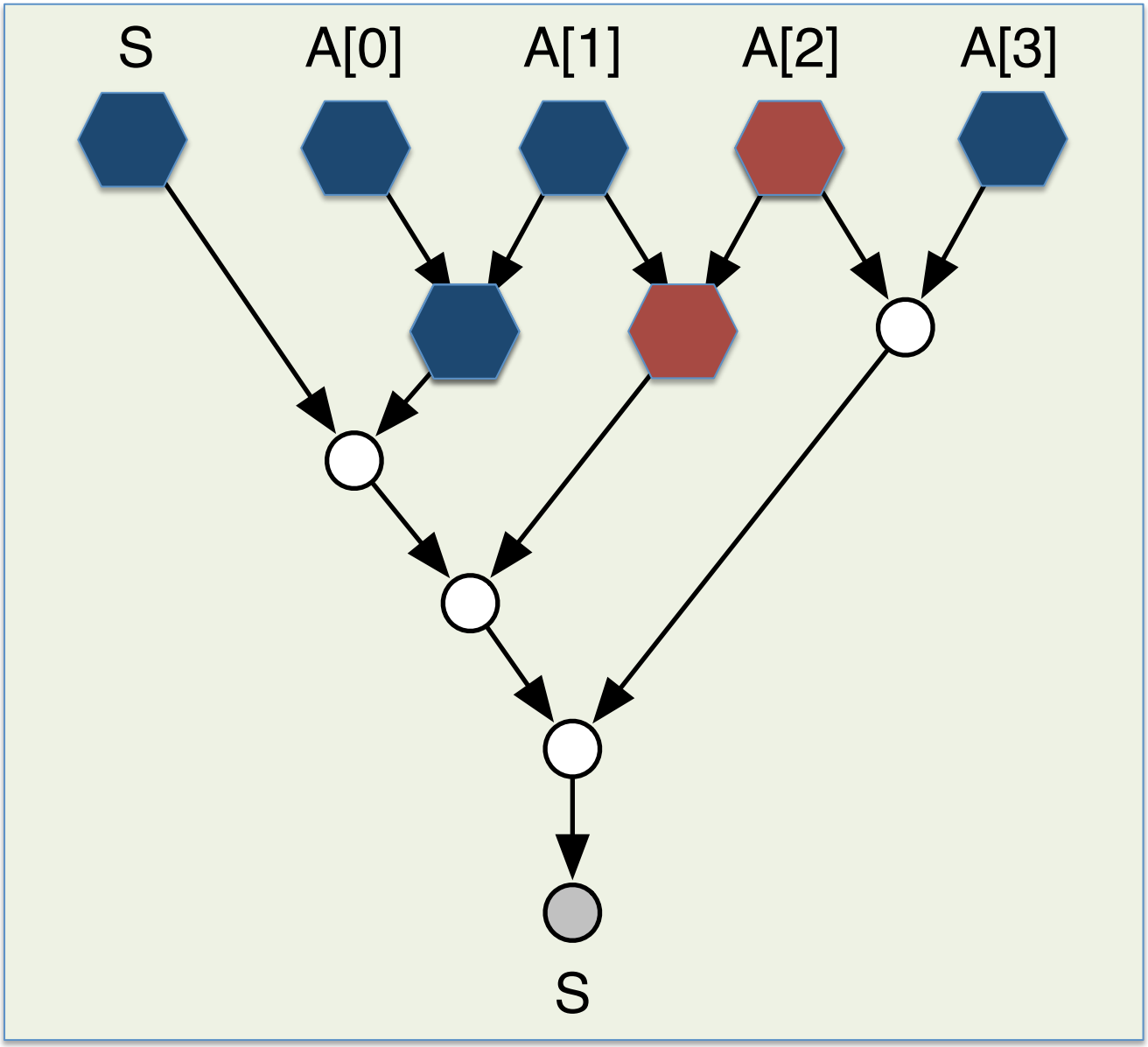
Red-Blue Pebble Game: 2 Red Pebbles



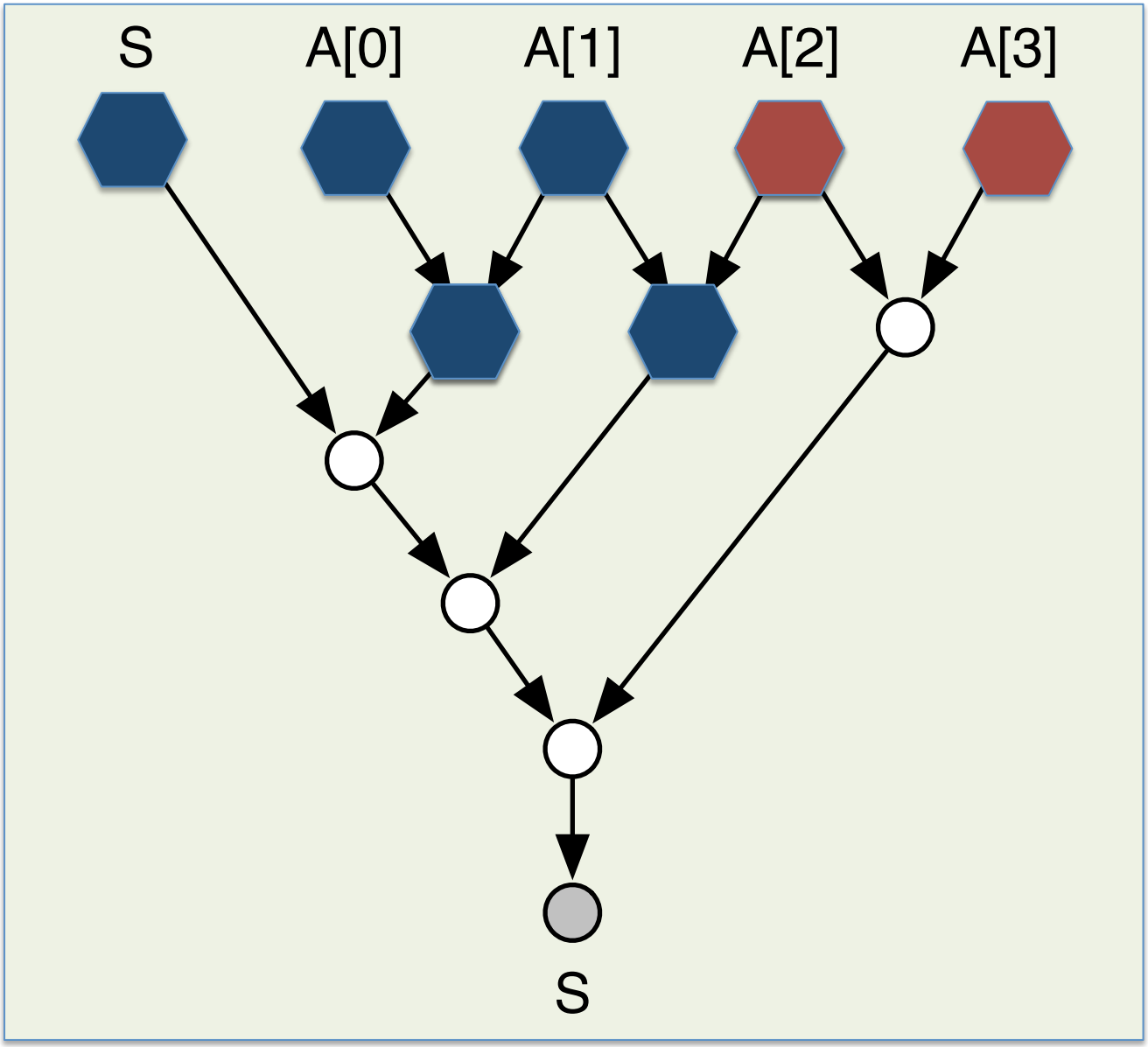
Red-Blue Pebble Game: 2 Red Pebbles



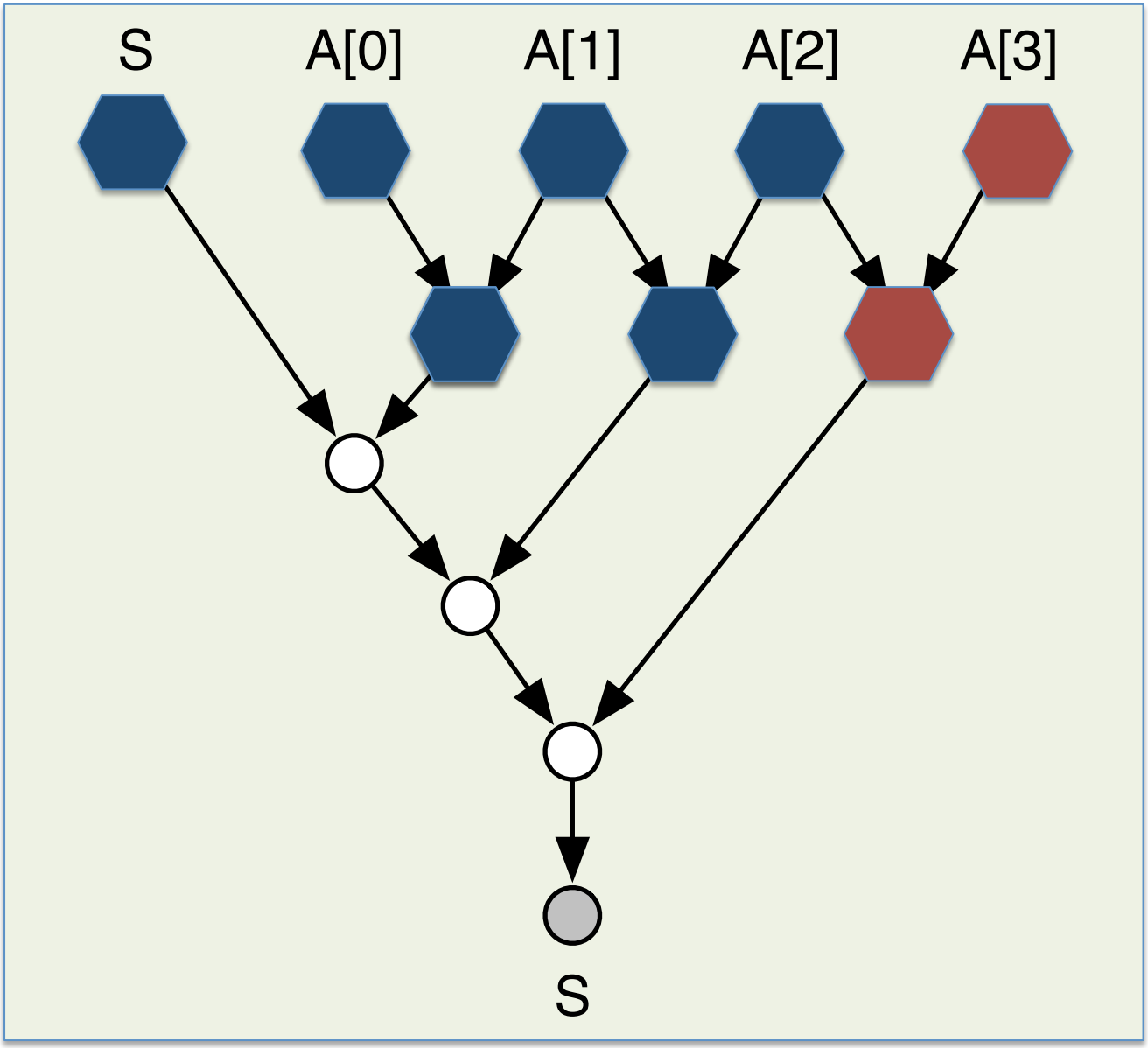
Red-Blue Pebble Game: 2 Red Pebbles



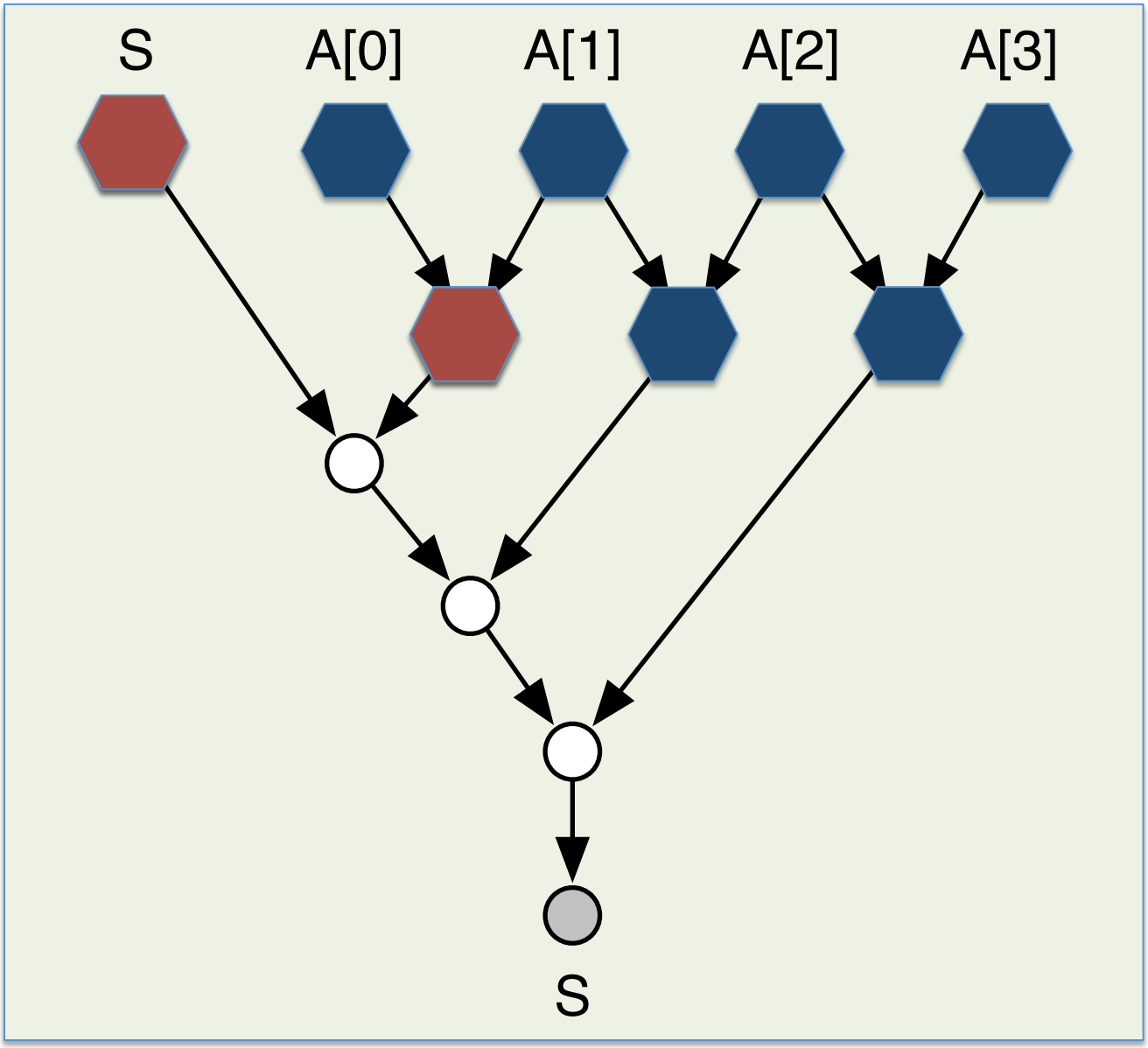
Red-Blue Pebble Game: 2 Red Pebbles



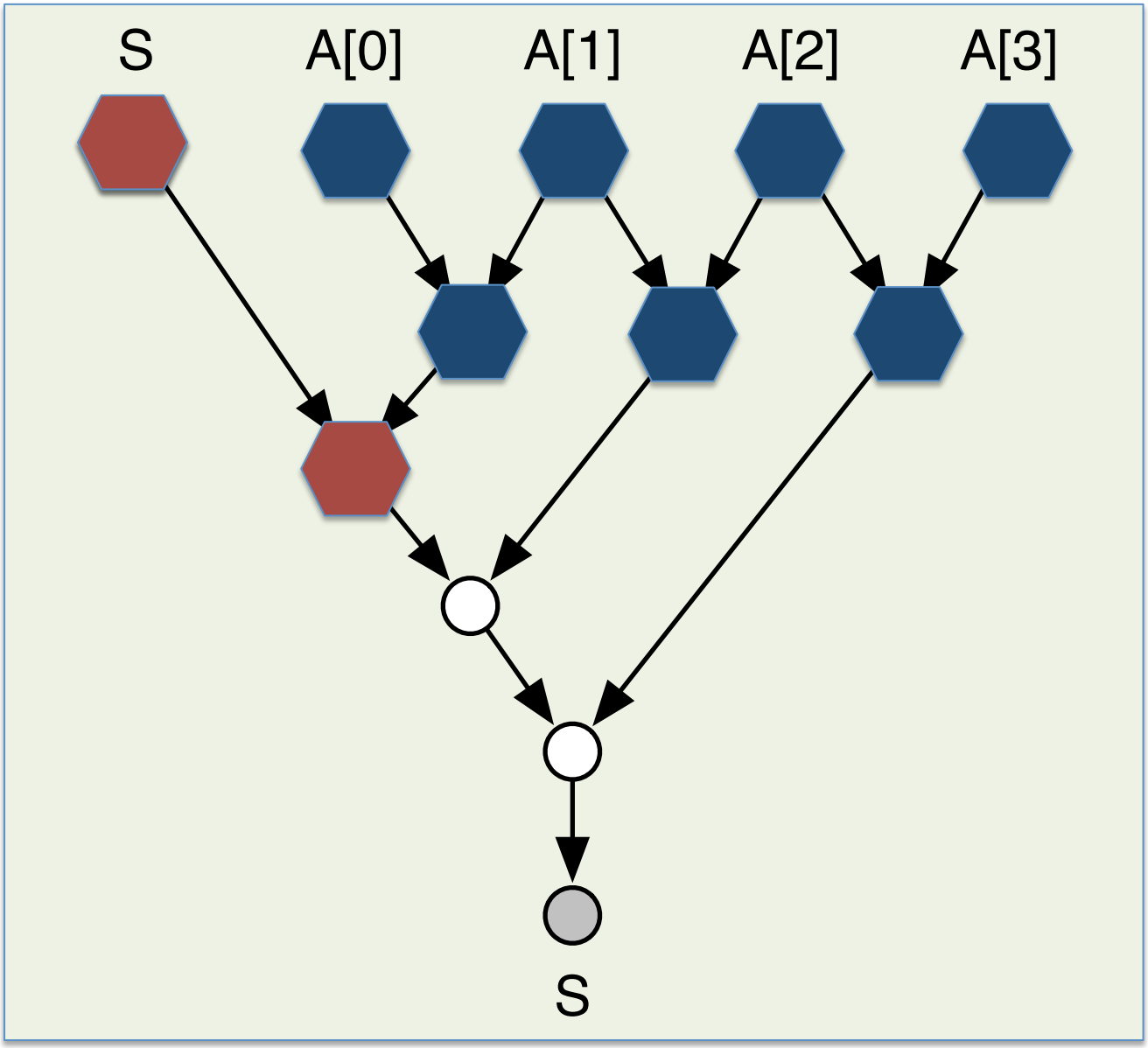
Red-Blue Pebble Game: 2 Red Pebbles



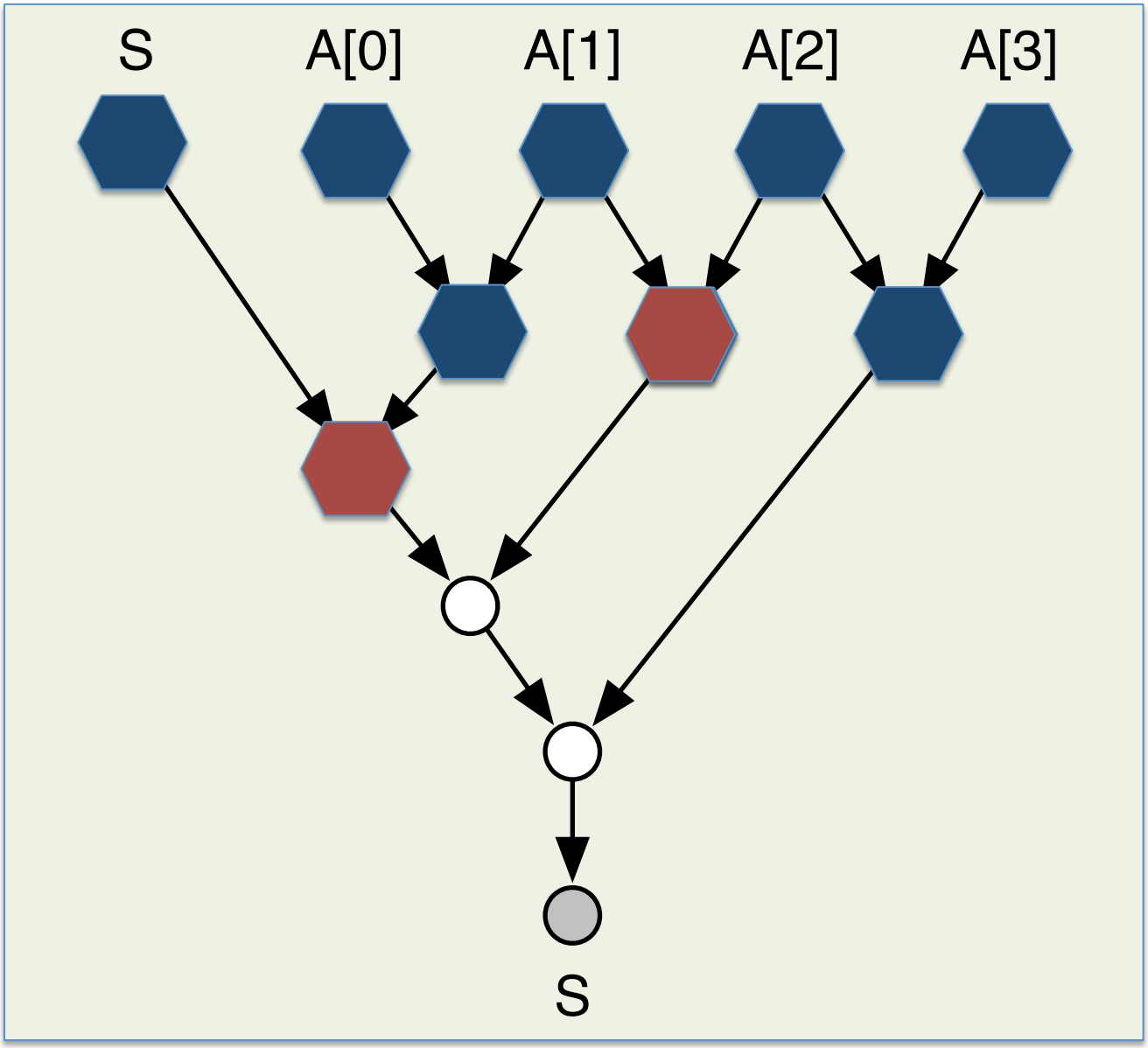
Red-Blue Pebble Game: 2 Red Pebbles



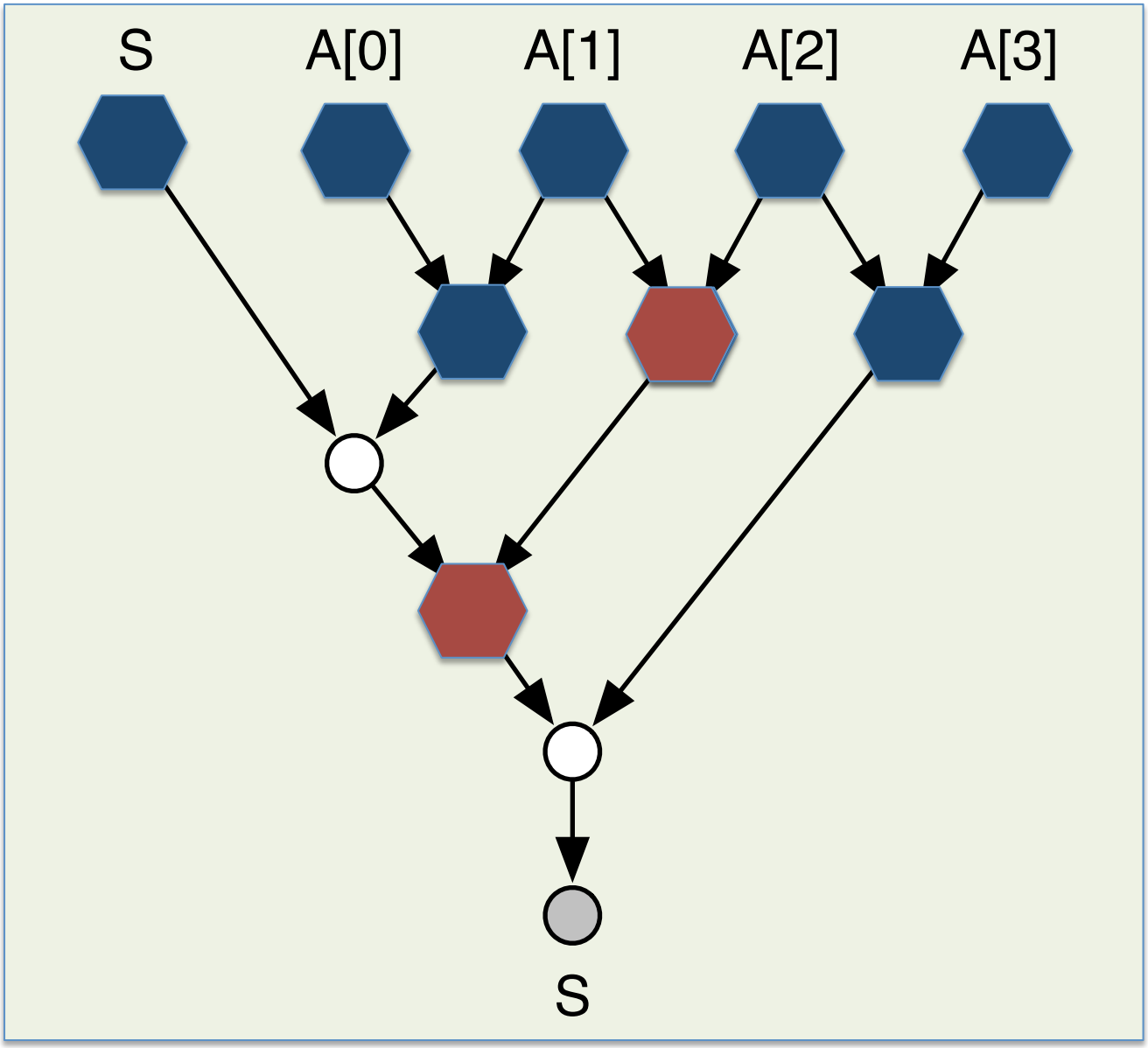
Red-Blue Pebble Game: 2 Red Pebbles



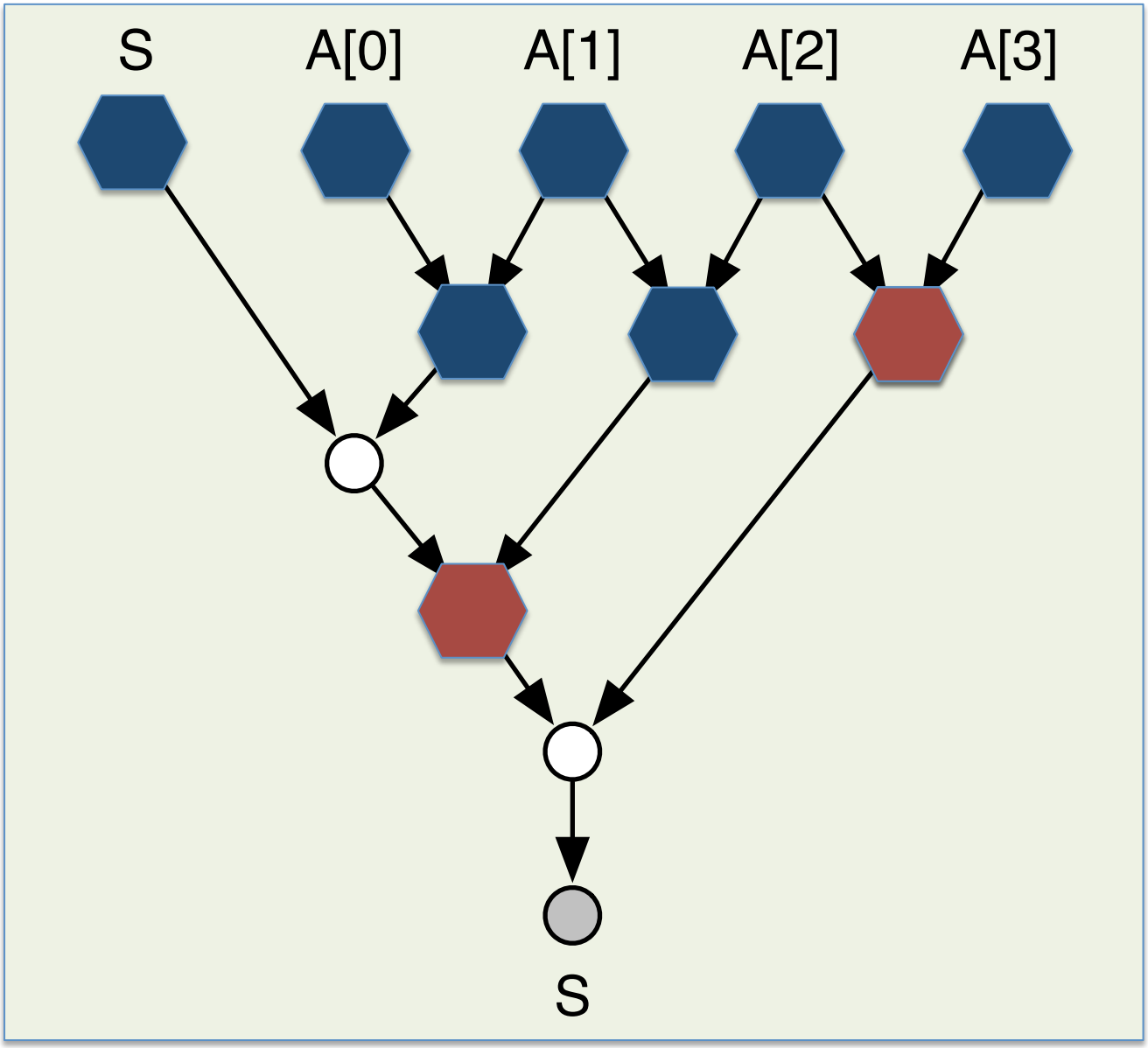
Red-Blue Pebble Game: 2 Red Pebbles



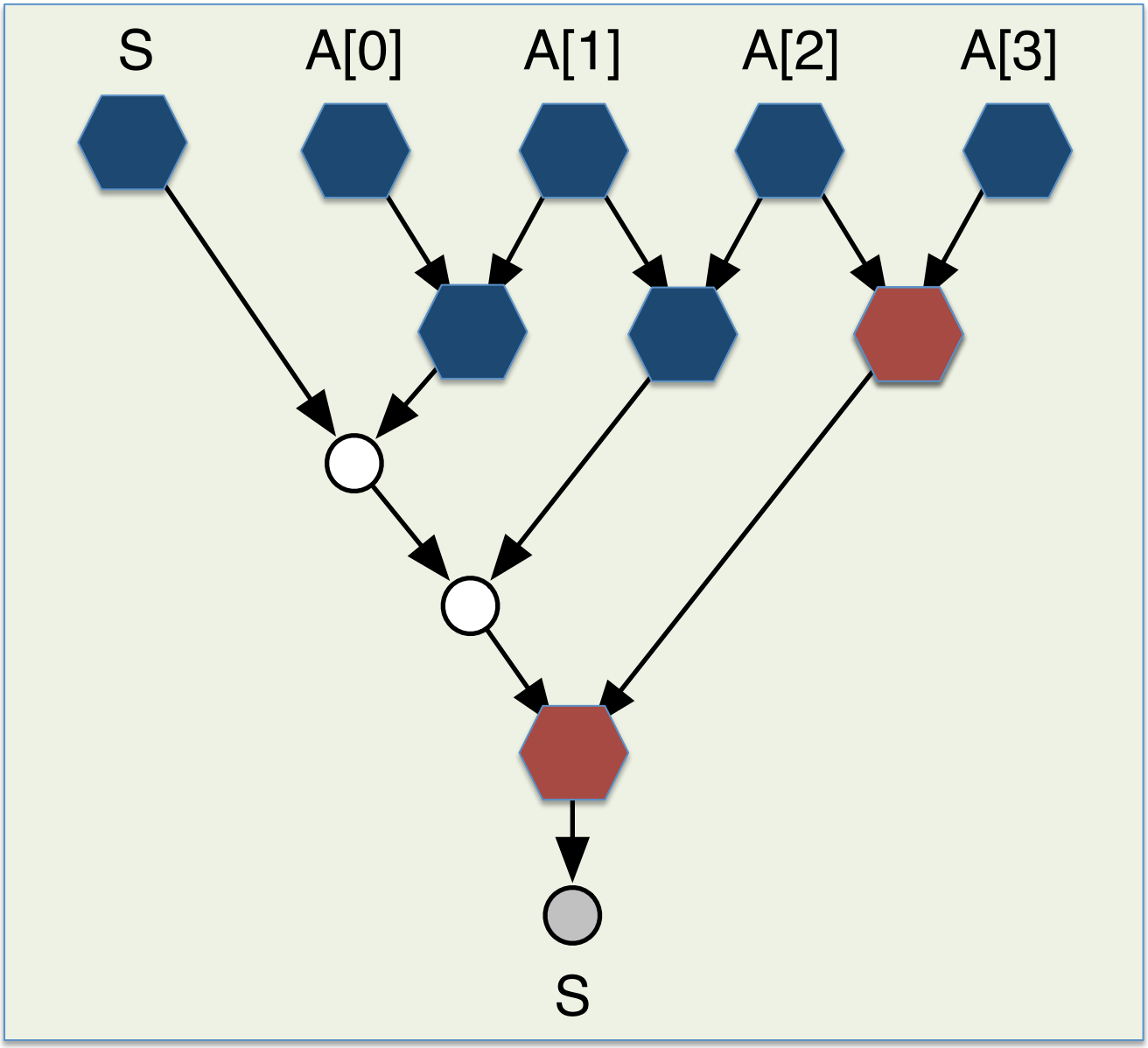
Red-Blue Pebble Game: 2 Red Pebbles



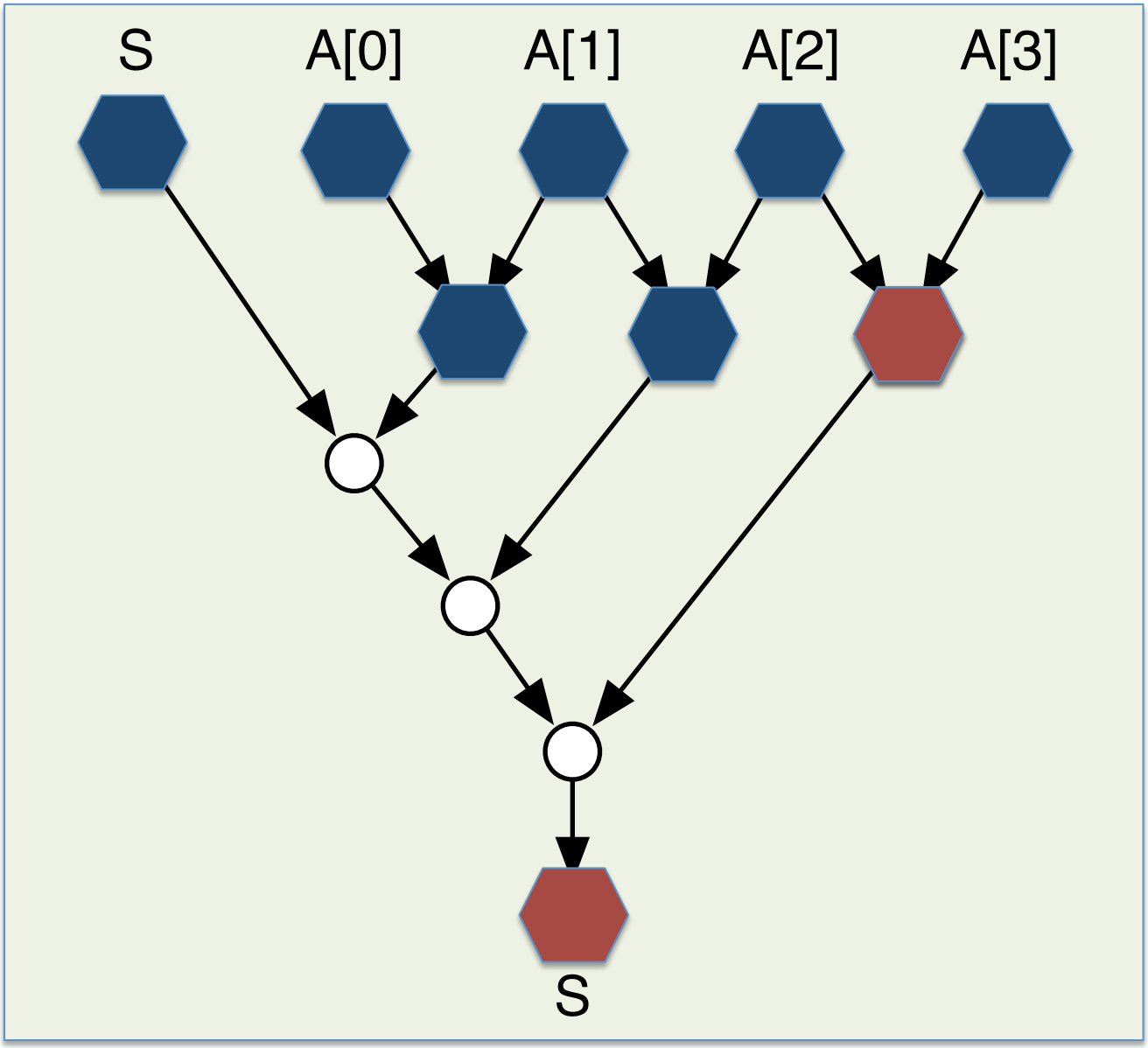
Red-Blue Pebble Game: 2 Red Pebbles



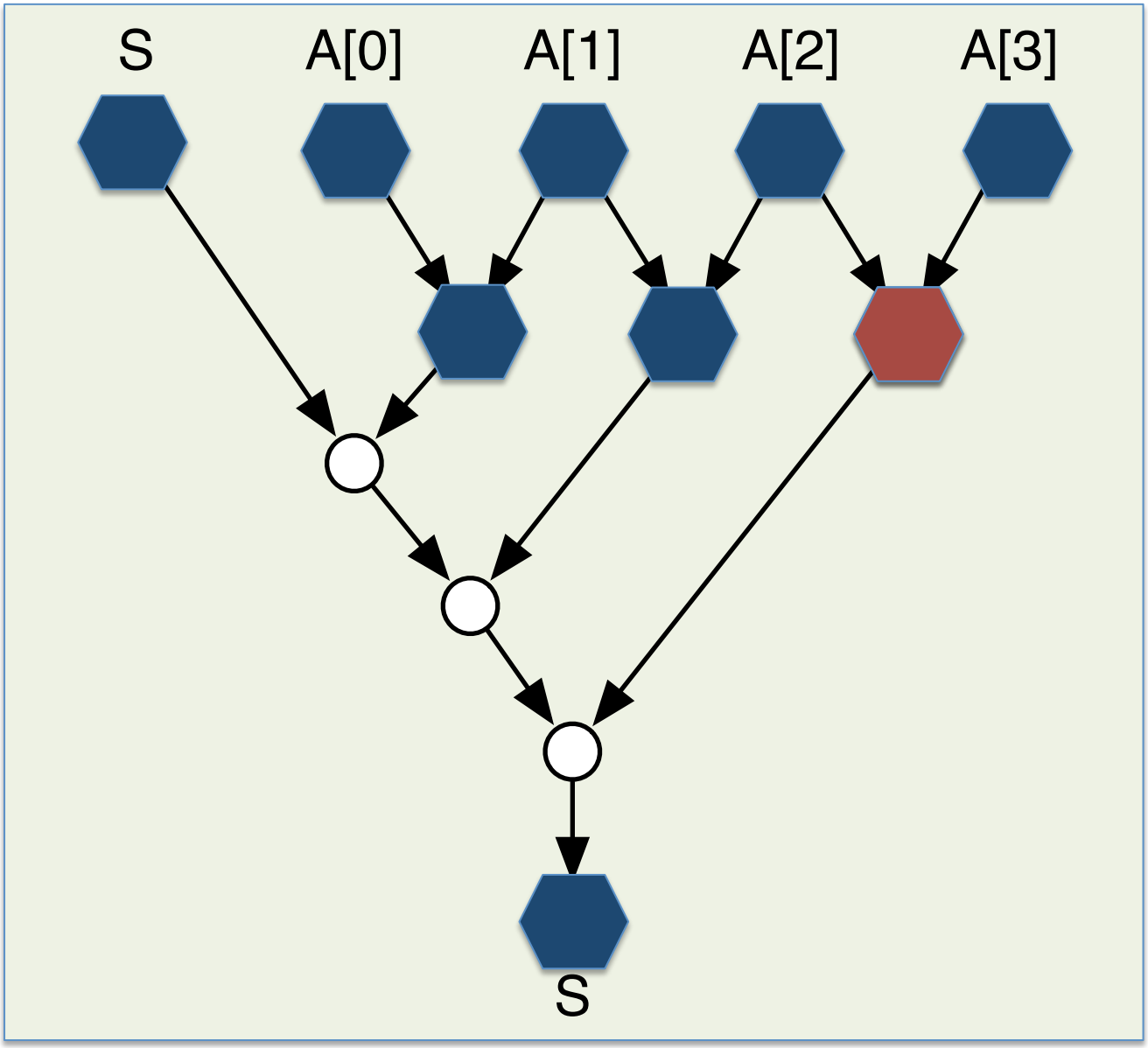
Red-Blue Pebble Game: 2 Red Pebbles



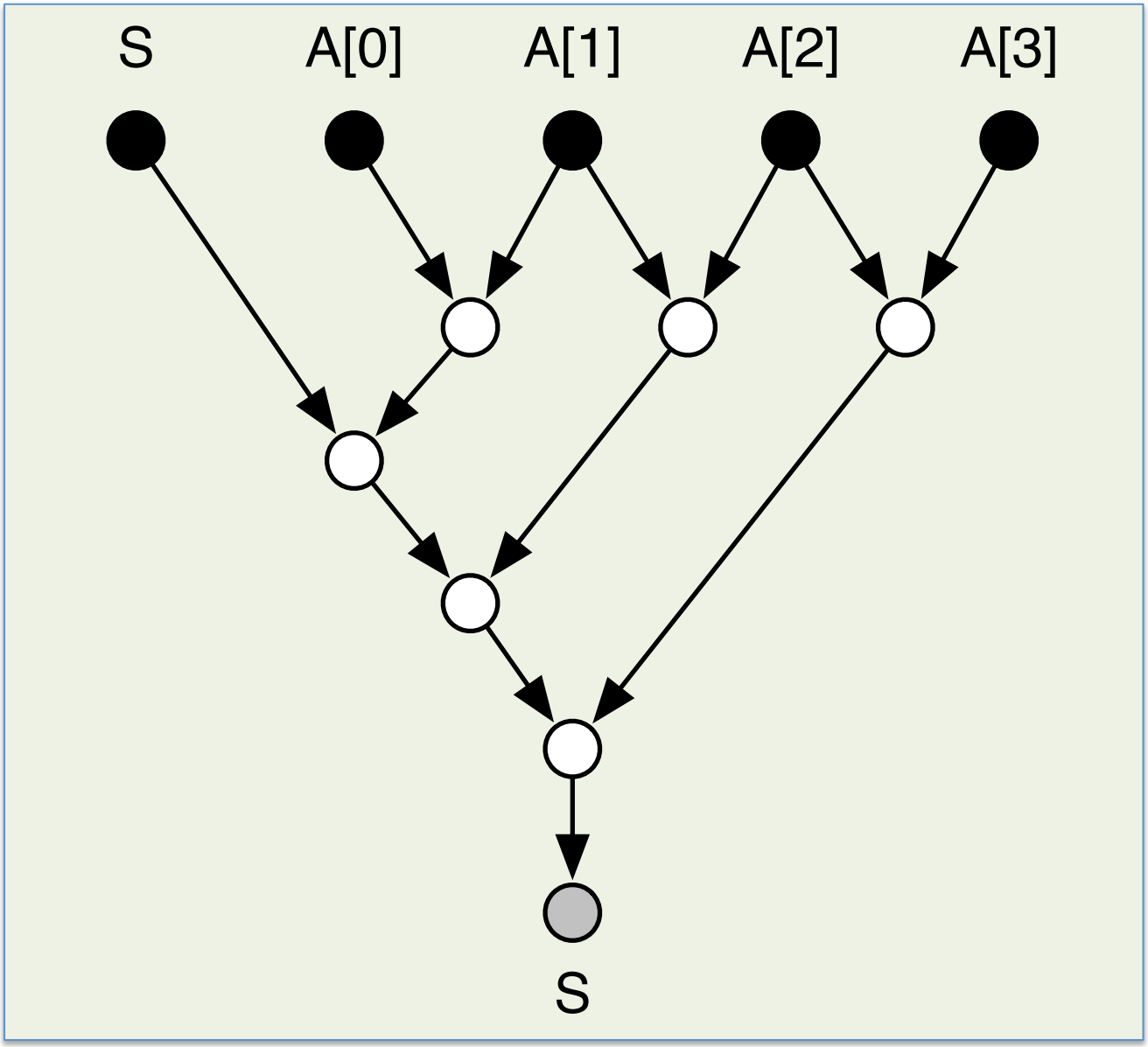
Red-Blue Pebble Game: 2 Red Pebbles



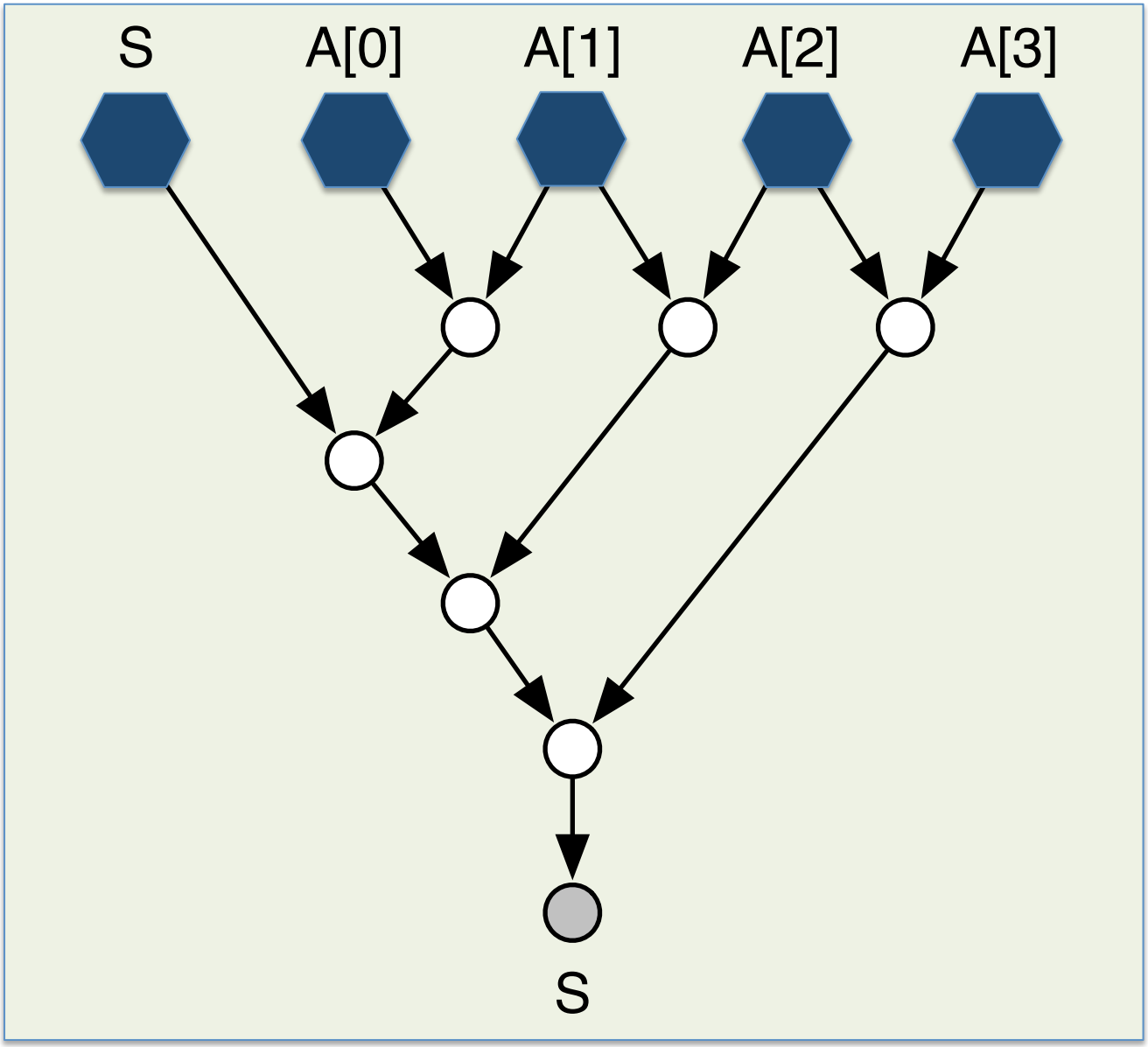
Red-Blue Pebble Game: 2 Red Pebbles



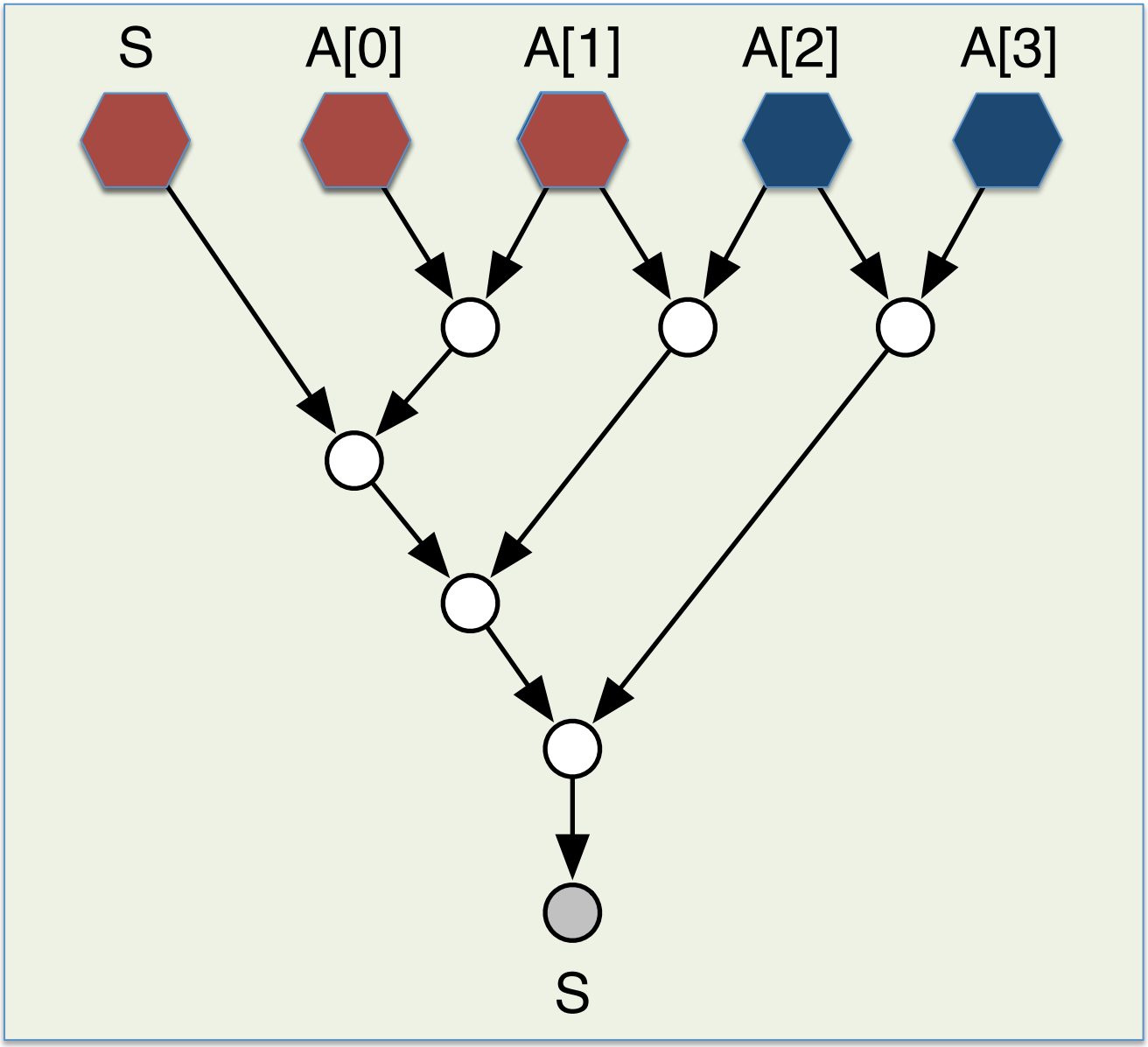
Red-Blue Pebble Game: 3 Red Pebbles



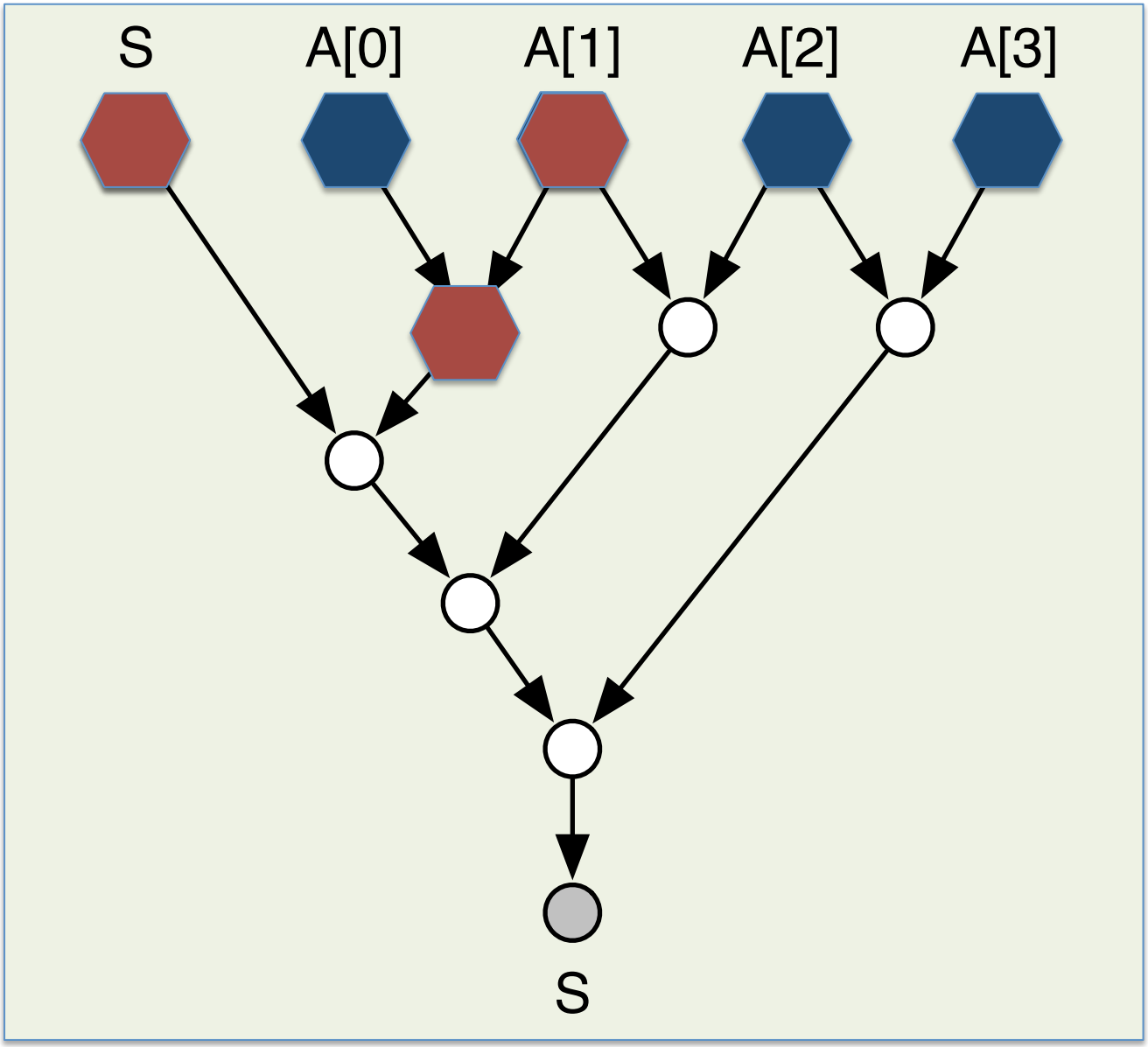
Red-Blue Pebble Game: 3 Red Pebbles



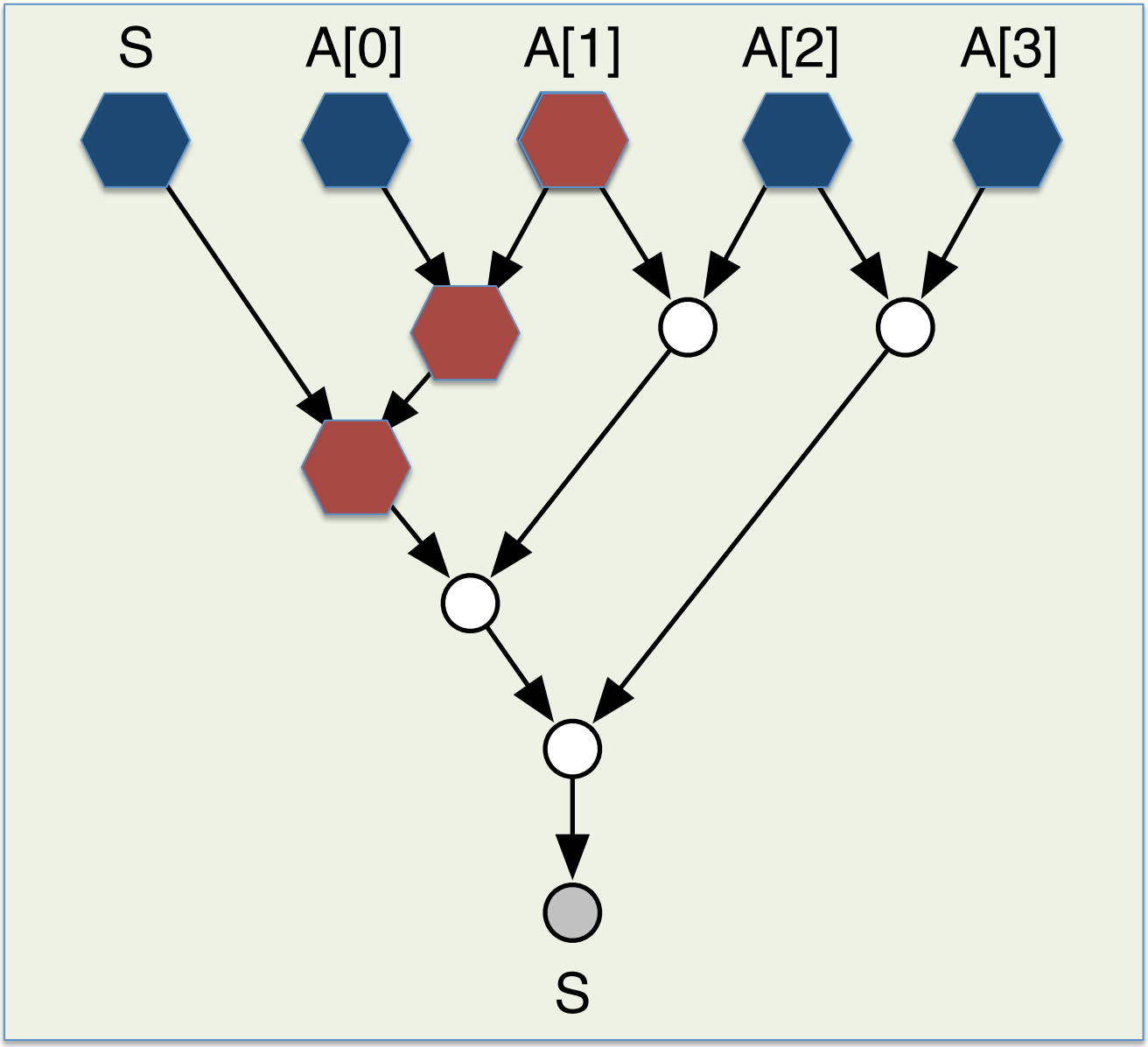
Red-Blue Pebble Game: 3 Red Pebbles



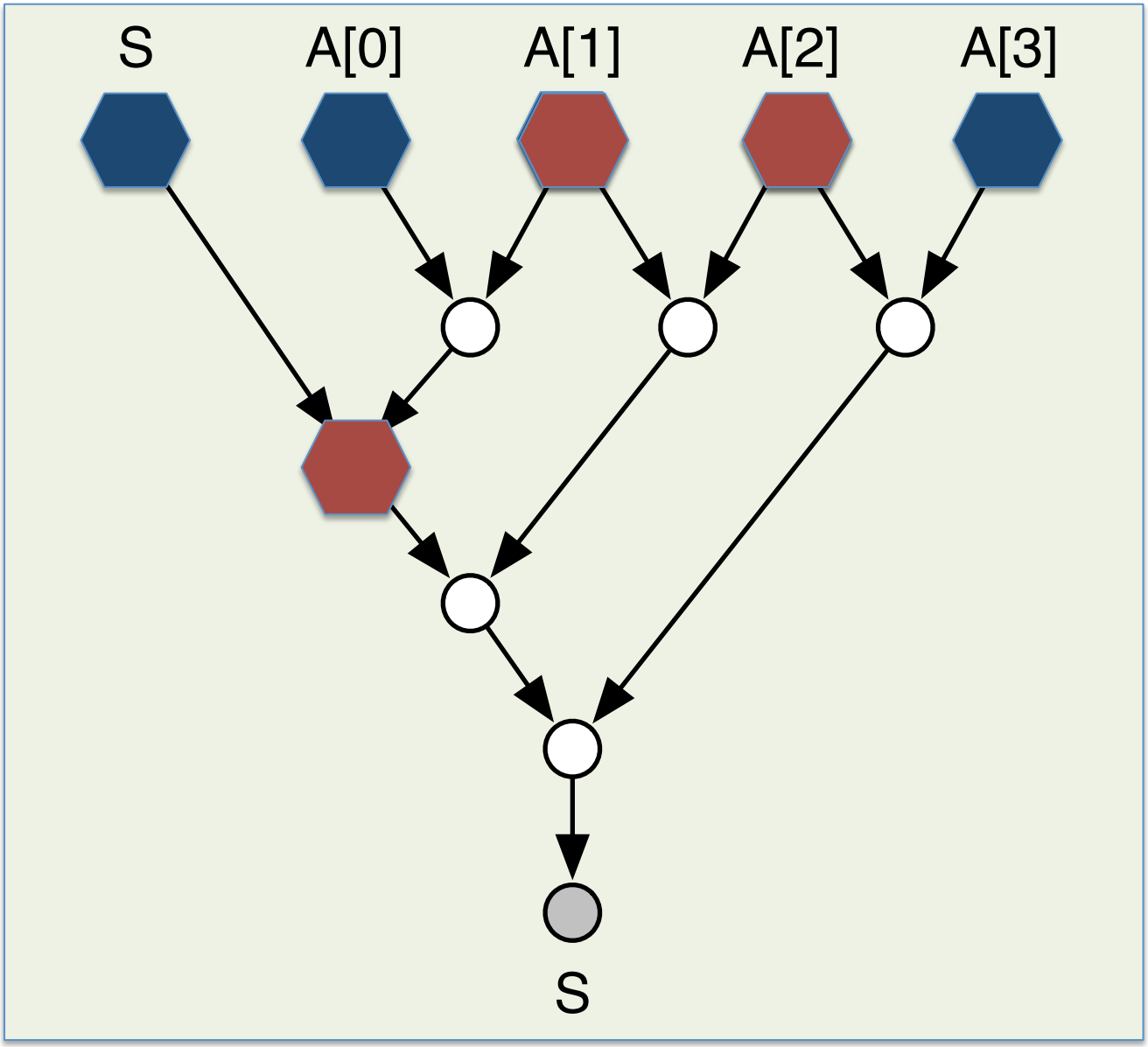
Red-Blue Pebble Game: 3 Red Pebbles



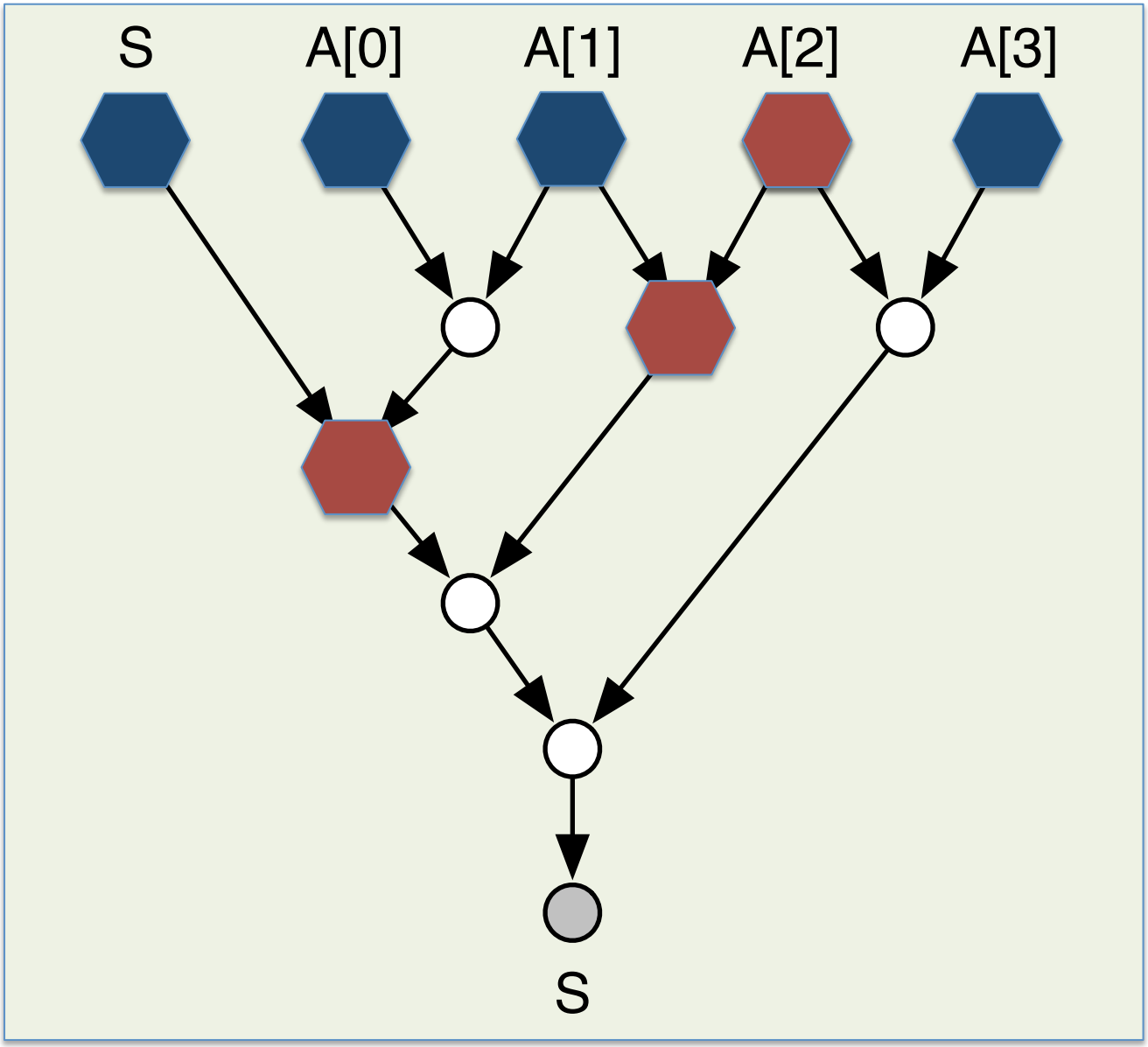
Red-Blue Pebble Game: 3 Red Pebbles



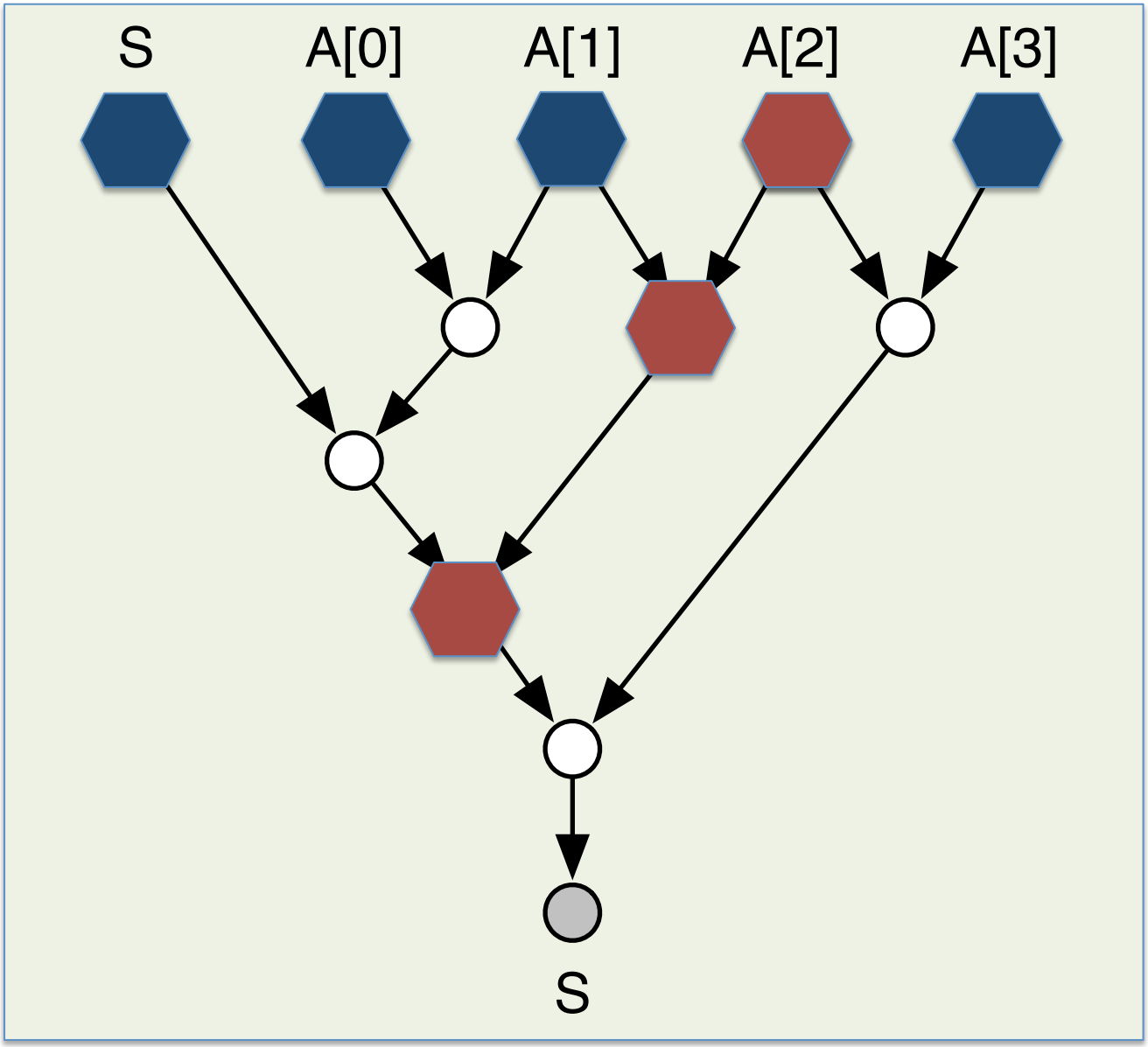
Red-Blue Pebble Game: 3 Red Pebbles



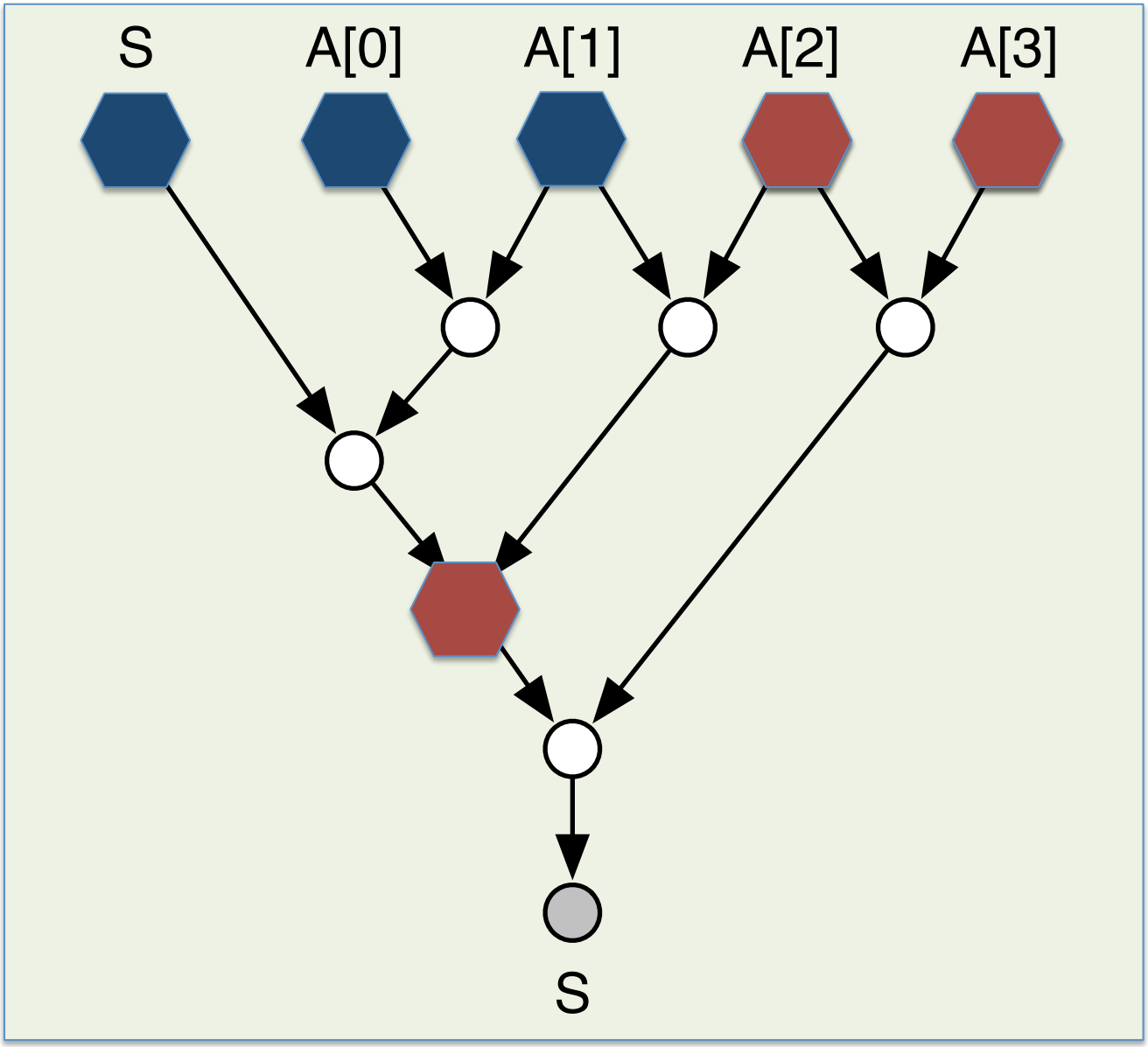
Red-Blue Pebble Game: 3 Red Pebbles



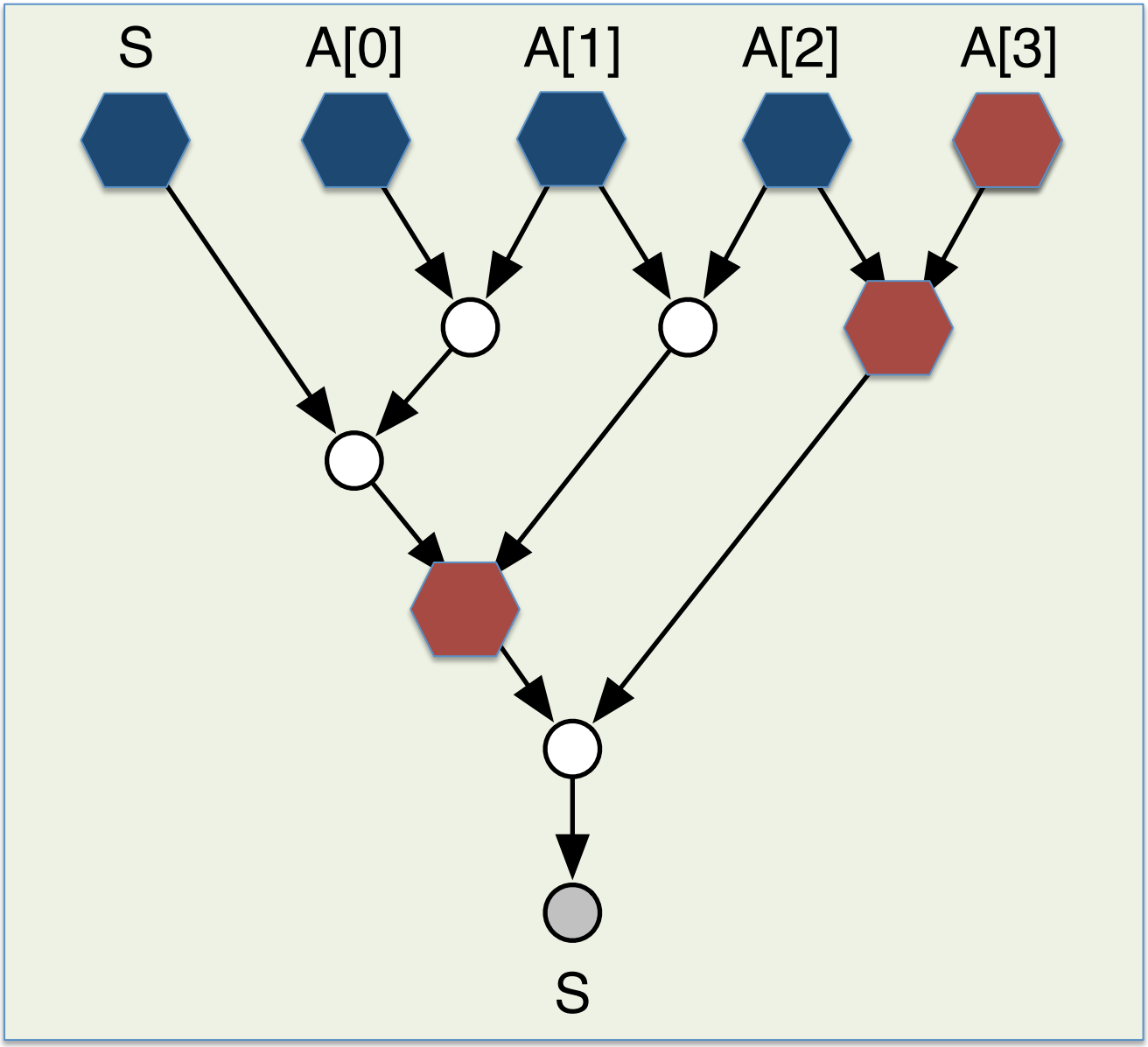
Red-Blue Pebble Game: 3 Red Pebbles



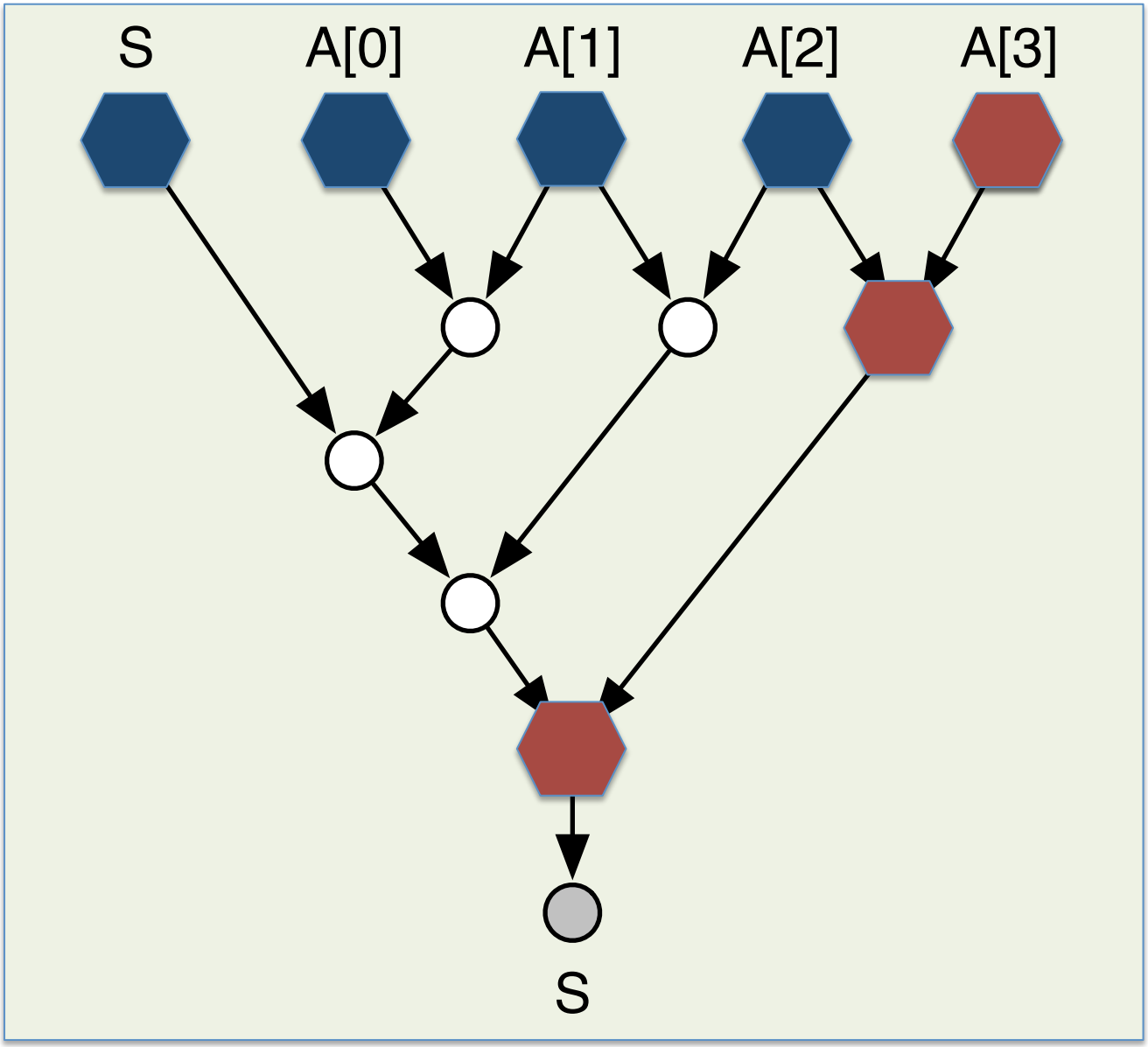
Red-Blue Pebble Game: 3 Red Pebbles



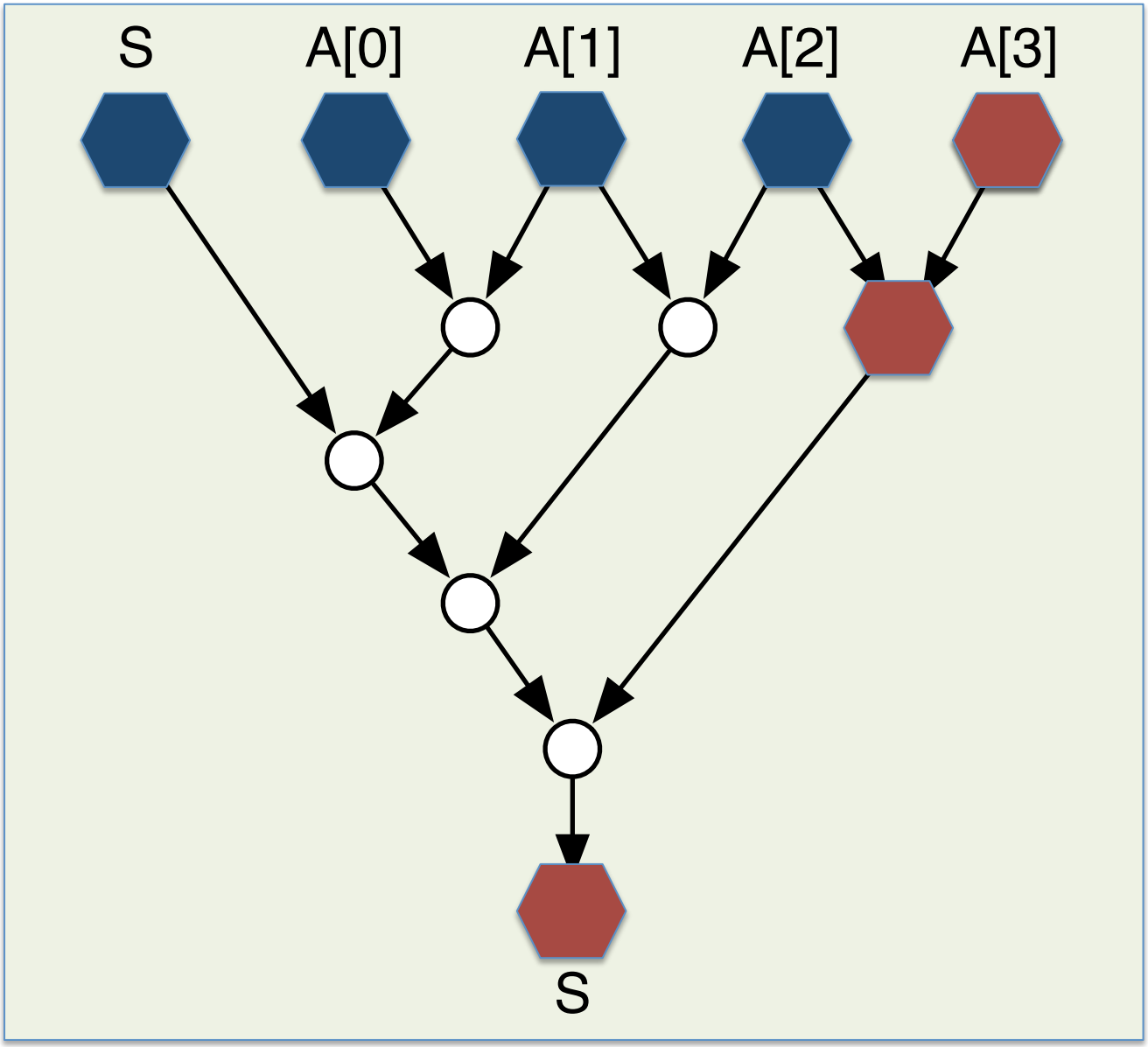
Red-Blue Pebble Game: 3 Red Pebbles



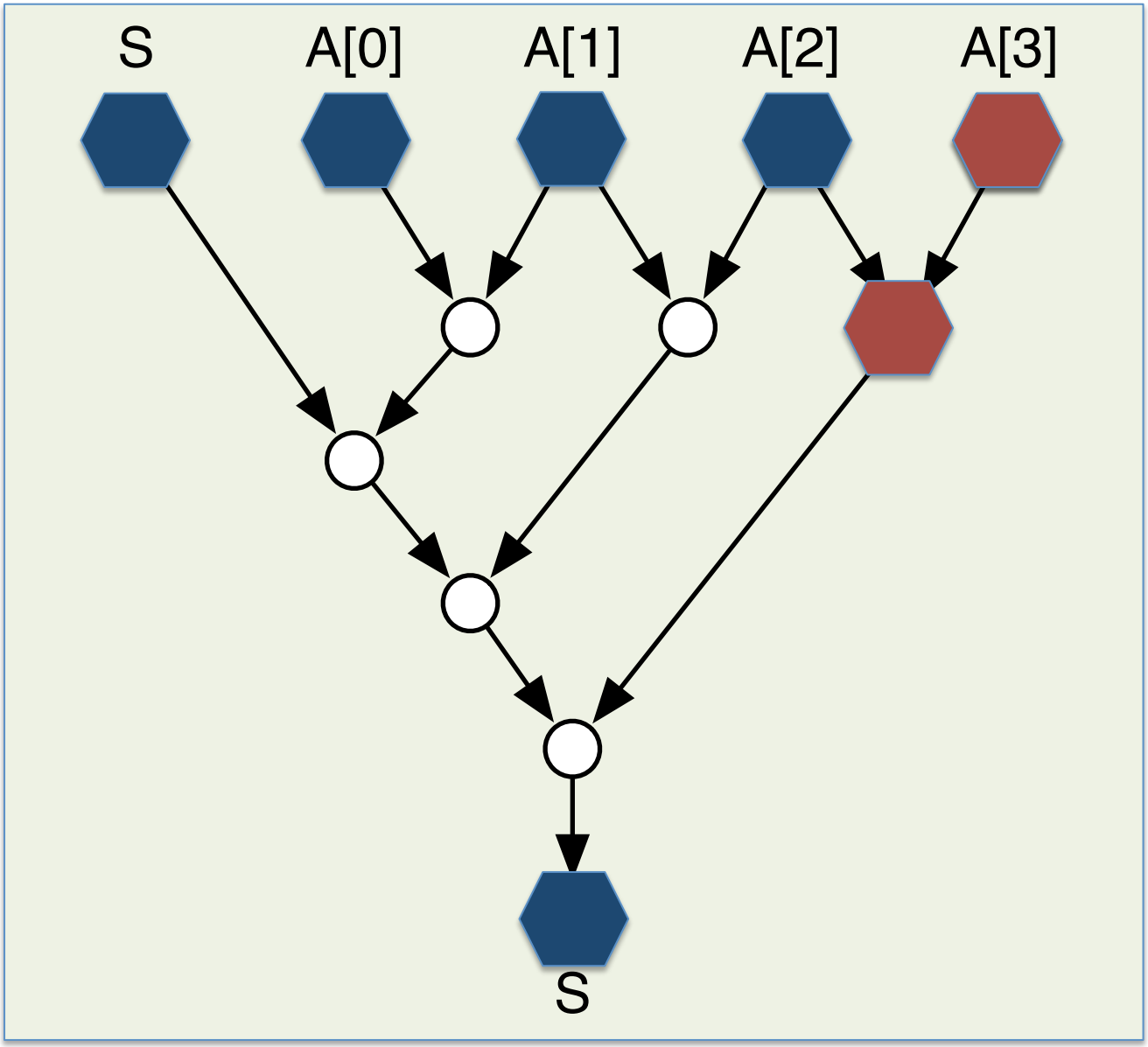
Red-Blue Pebble Game: 3 Red Pebbles



Red-Blue Pebble Game: 3 Red Pebbles



Red-Blue Pebble Game: 3 Red Pebbles



Hong-Kung (1981): S-partitioning of CDAG

Given a CDAG G. An S-partitioning of G is a collection of h subsets of $\mathcal{V}$ such that:

P1 $\bigcap_{i=1}^{h} \mathcal{V}_i = \emptyset$, and $\bigcup_{i=1}^{h} \mathcal{V}_i = \mathcal{V}$

P2 there is no circuit between subsets

P3 $\forall i, \quad \exists D \in \mathsf{Dom}(\mathcal{V}_i) \quad$ such that $\quad |D| \leq S$

P4 $\forall i, \quad |\mathsf{Min}(\mathcal{V} f_i)| \leq S$

Where

- a dominator set of $\mathcal{V}_i$, $D \in \mathsf{Dom}(\mathcal{V}_i)$ is a set of vertices such that any path from I to a vertex in $\mathcal{V}_i$ contains some vertex in D;
- the minimum set of $\mathcal{V}_i$, $\mathsf{Min}(\mathcal{V}_i)$ is the set of vertices in $\mathcal{V}_i$ that have at least one child outside of $\mathcal{V}_i$; and $|Set|$ is the cardinality of the set Set.

Hong-Kung (1981): S-partitioning theorem

Any complete calculation of the red-blue pebble game on on a CDAG using at most S red pebbles, is associated with a $2S$-partition of CDAG such that

$$S \times h \geq q \geq S \times (h - 1)$$

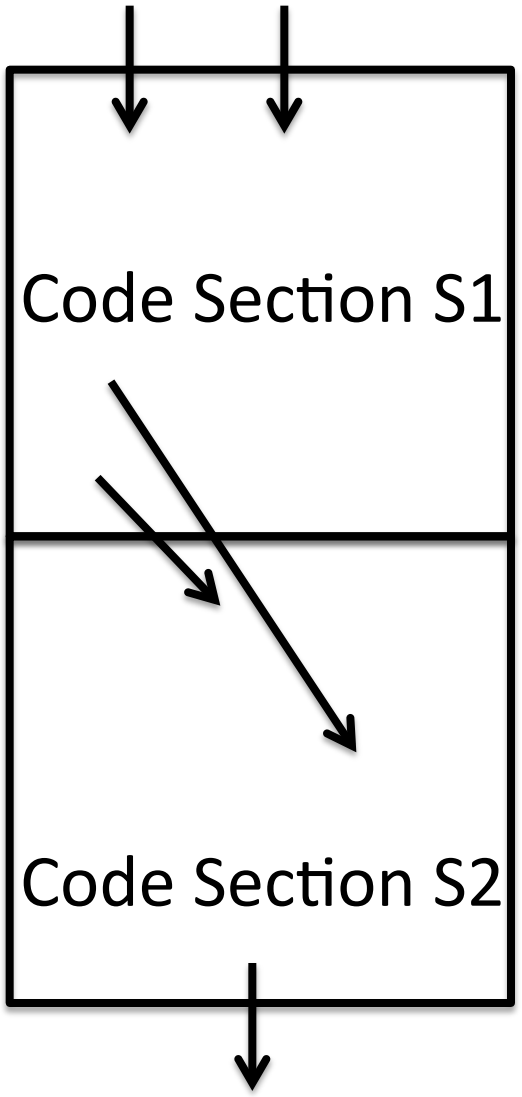
where q is the number of I/O moves in the game and h is the number of subsets in the $2S$-partition.

Lower Bound

Let $H(2S)$ be the minimal number of vertex sets for any valid $2S$-partition of a given CDAG. Then the minimal number Q of I/O operations for any valid execution of the CDAG is bounded by:

$$Q \geq S \times (H(2S) - 1)$$

The Composability Problem



- The IO for a computation includes all inputs and outputs of the CDAG
- Consider S = S1 ; S2
- If some outputs of S1 are inputs to S2, they may be passed in red pebbles in a complete game for S
- OptIO(S) <= OPtIO(S1)+OPtIO(S2)
- Hence Lbound(S1) and Lbound(S2) cannot simply be added to get Lbound(S)
- In contrast Comp(S) = Comp(S1)+Comp(S2)

Addressing Composability

1. Modify pebble game to disallow re-computation
   - Add a third type of pebble: white
   - Placed white pebble on a vertex, along with red pebble, when vertex is first computed
   - Compute rule changed to only allow it when vertex does not have a white pebble (no change to load, store and delete rules)
2. Define a new metric IIO (internal IO) of CDAG G:
   - Given G = (I,V,E,O), internal CDAG G' = (0,V,E,0)
   - IIO(G) is IO(G'), for optimal game on G'
   - $IIO(G) \leq IO(G) \leq IIO(G)+|I|+|O|$

Composable Lower Bounds

$IIO(G1)+IIO(G2) \leq IIO(G1 \cup G2)$

where as

$IO(G1 \cup G2) \leq IO(G1)+IO(G2)$

Lower Bounds NOT composable

Red-Blue-White Pebble Game

Given a CDAG $G = (I, V, E, O)$, S red pebbles and an arbitrary number of blue and white pebbles, with a blue and white pebble on each *input* vertex, a complete game is any sequence of steps obeying the following rules that results a final state with blue pebbles on all *output* vertices:

R1 (Input) A red pebble may be placed on any vertex that has a blue pebble,

R2 (Output) A blue pebble may be placed on any vertex that has a red pebble,

R3 (Compute) If a vertex v does not have a white pebble and all its immediate predecessors have red pebbles on them, a red pebble along with a white pebble may be placed on v.

R4 (Delete) A red pebble may be removed from any vertex (reuse storage).

S-partitioning of CDAG – new definition

Given a CDAG G. An S-partitioning of G is a collection of h subsets of $\mathcal{V} - I$ such that:

P1 $\bigcap_{i=1}^{h} V_i = \emptyset$, and $\bigcup_{i=1}^{h} V_i = \mathcal{V} - I$

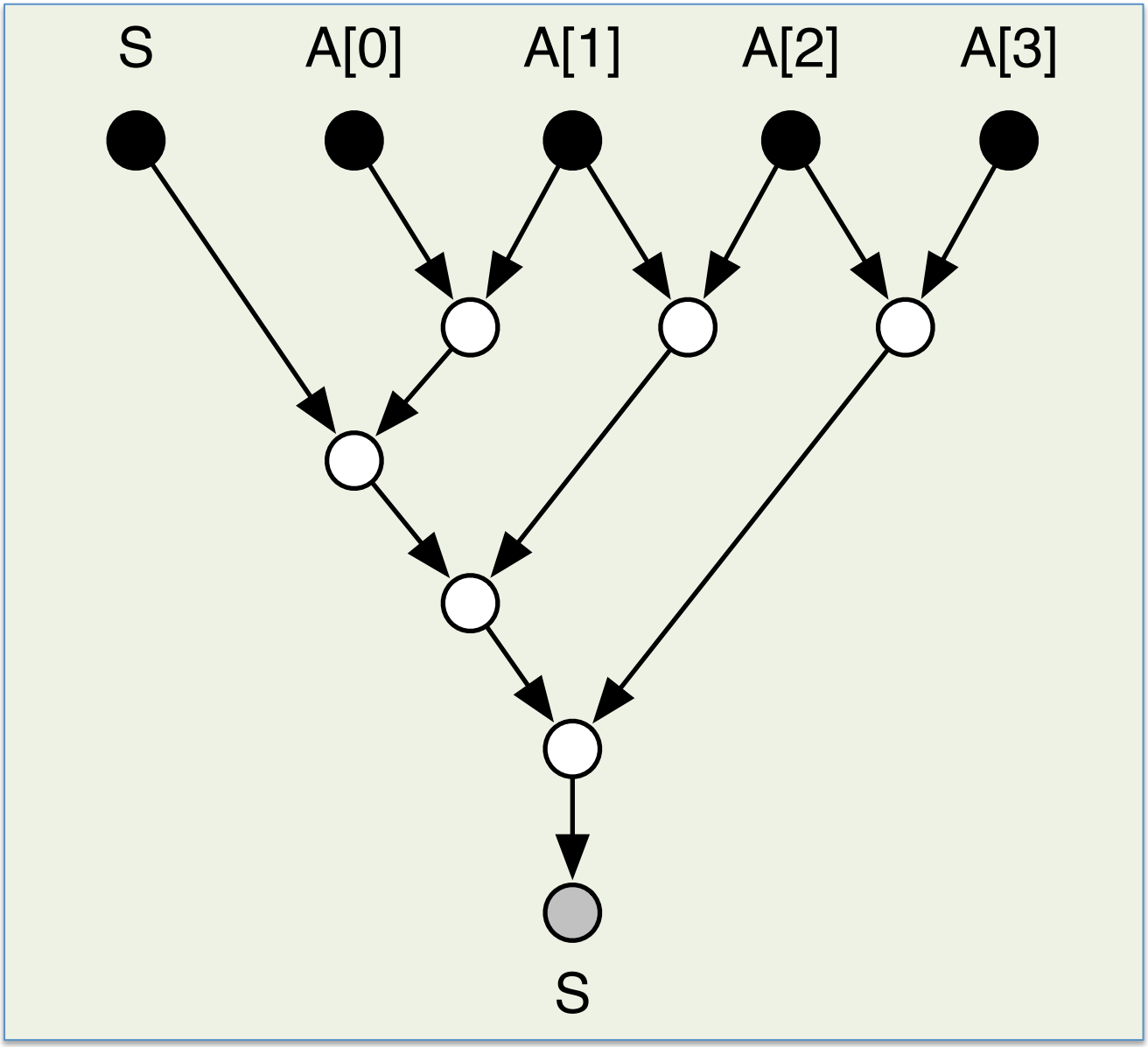
P2 there is no circuit between subsets

P3 $\forall i, \quad |\mathsf{In}(V_i)| \leq S$

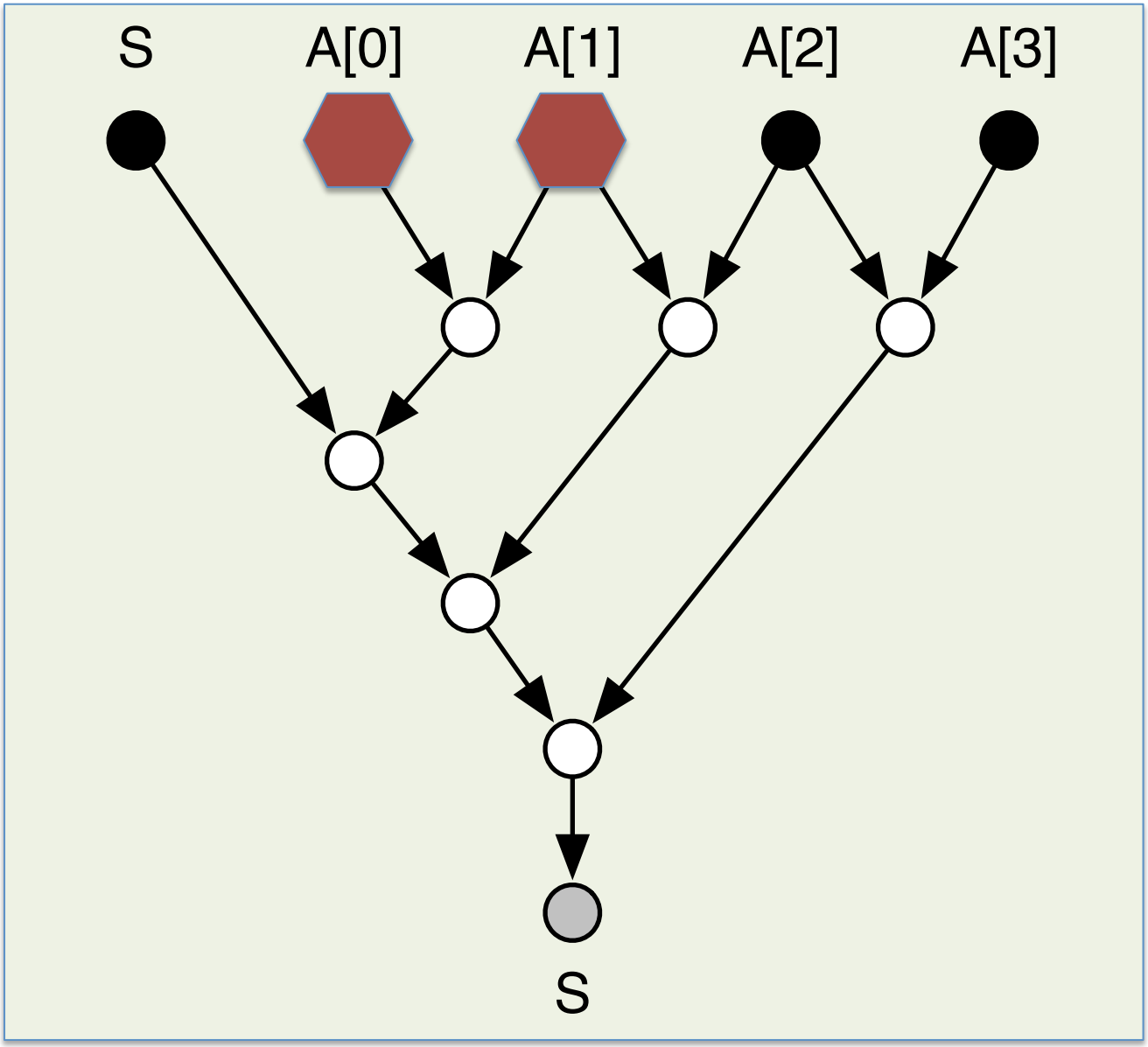
P4 $\forall i, \quad |\mathsf{Out}(V_i)| \leq S$

Where the input set of V_i, $\mathsf{In}(V_i)$ is the set of vertices of $\mathcal{V} - V_i$ that have at least one child in V_i; the output set of V_i, $\mathsf{Out}(V_i)$ is the set of vertices of V_i that have at least one child outside of $\mathcal{V}_i$; and $|Set|$ is the cardinality of the set Set.

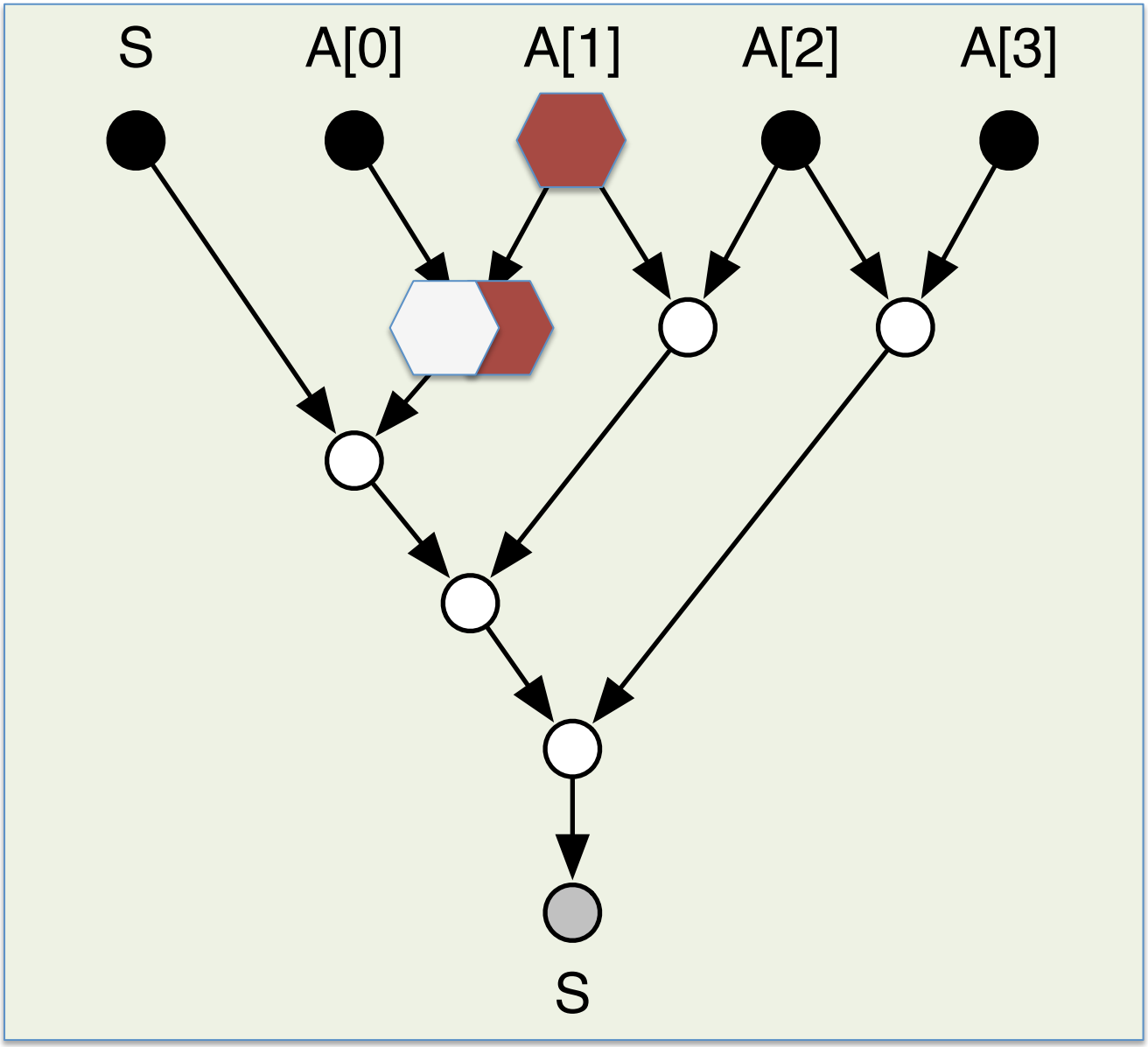
IIO RBW Pebble Game: 2 Red Pebbles



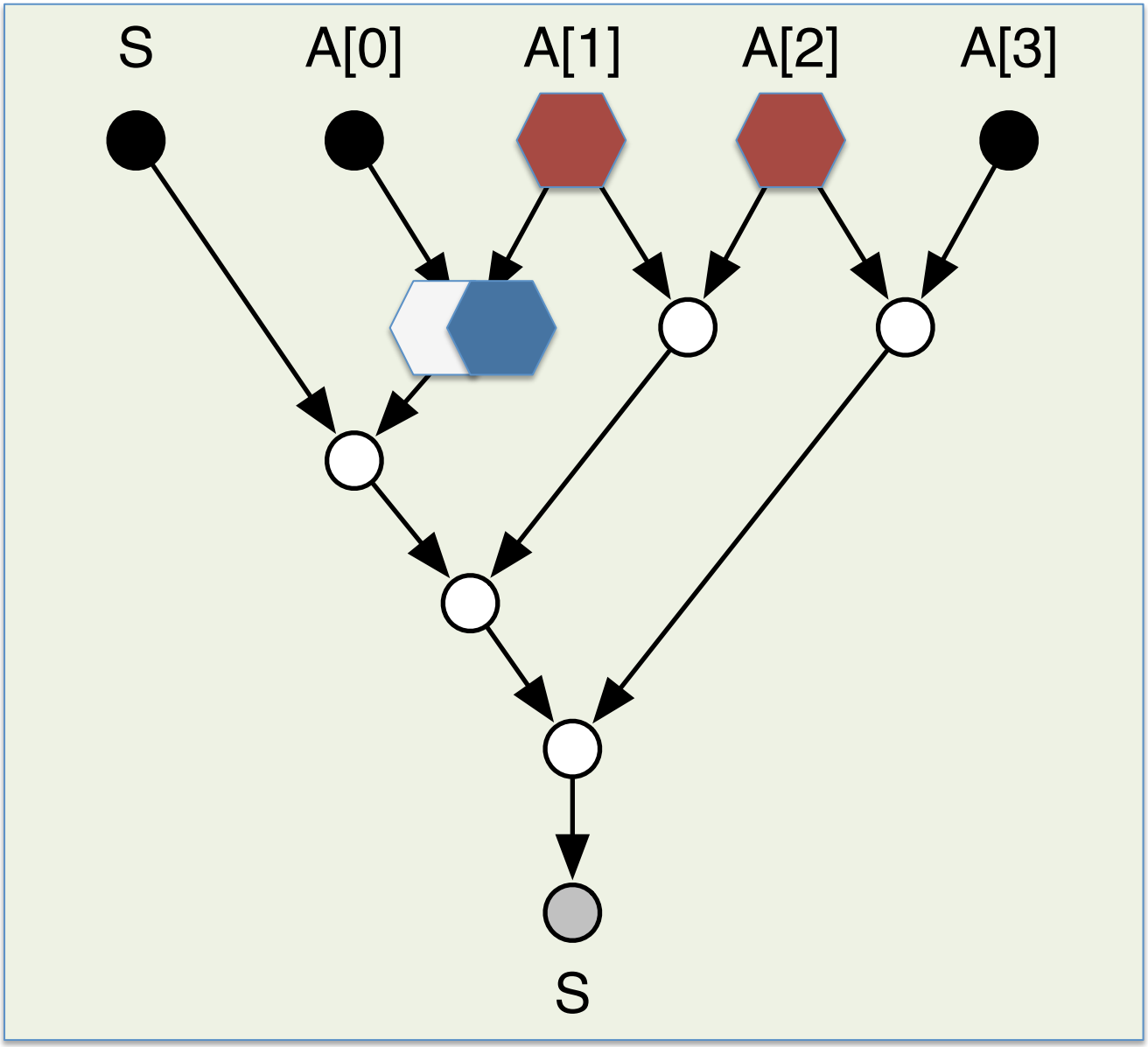
IIO RBW Pebble Game: 2 Red Pebbles



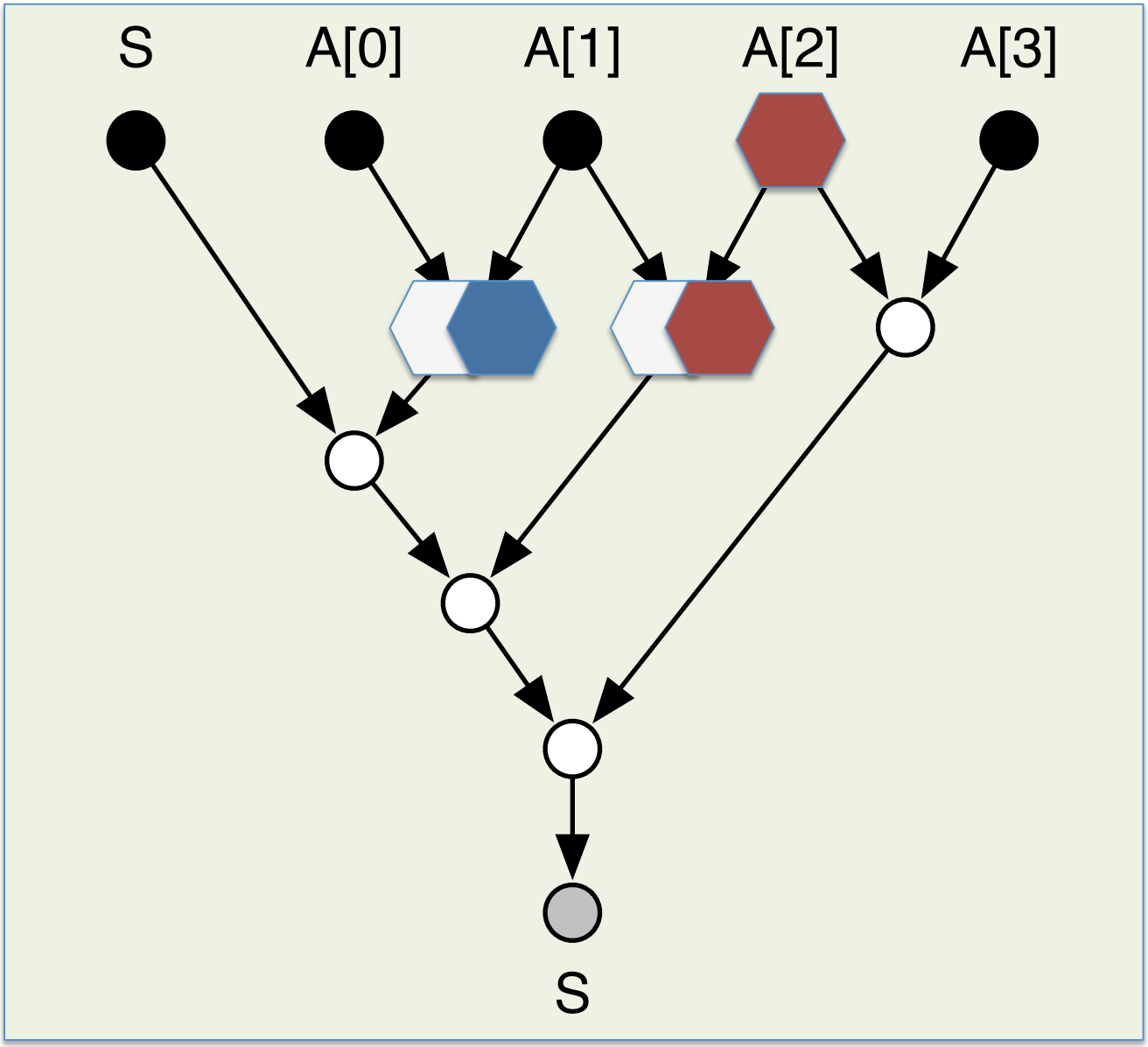
IIO RBW Pebble Game: 2 Red Pebbles



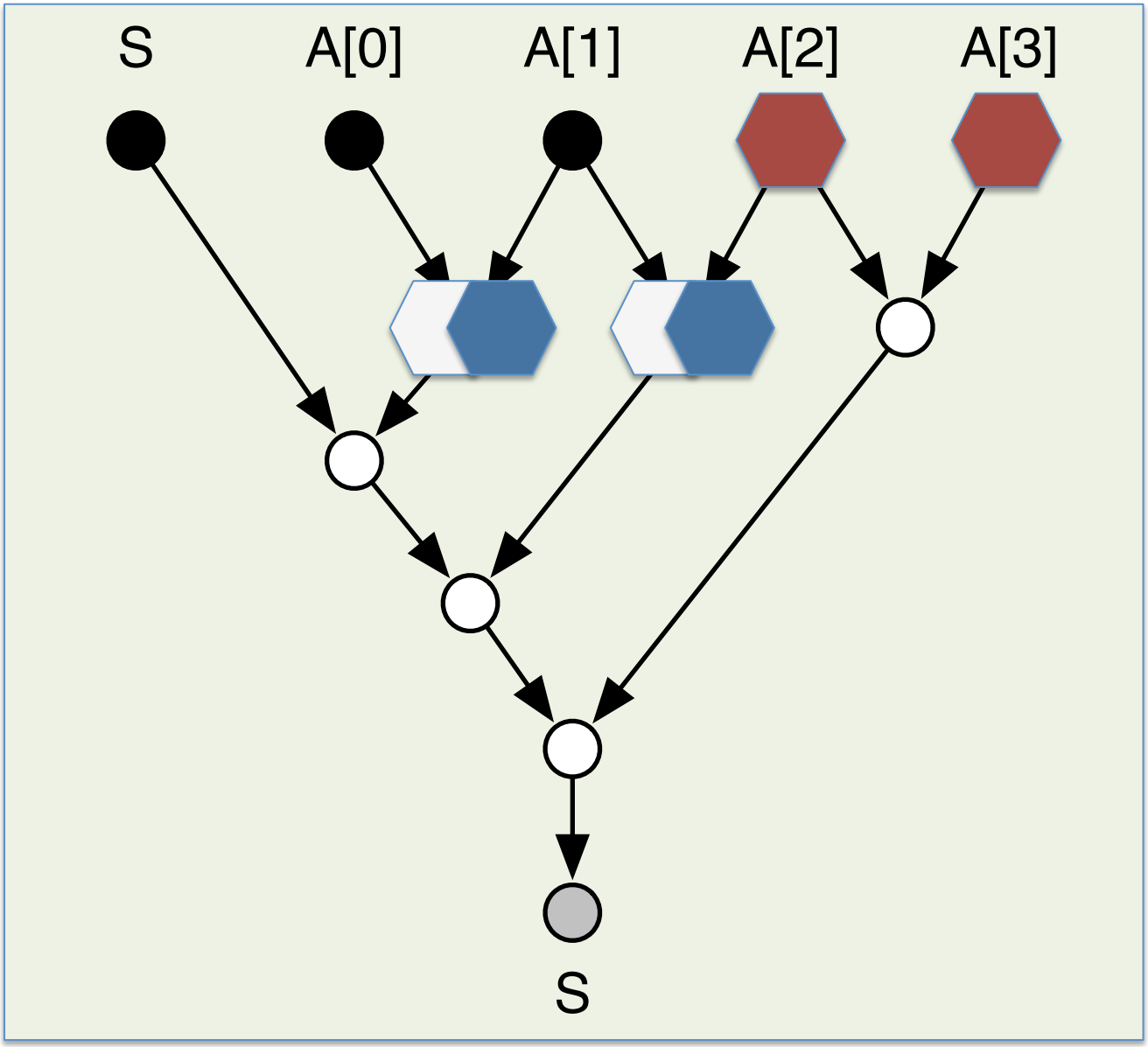
IIO RBW Pebble Game: 2 Red Pebbles



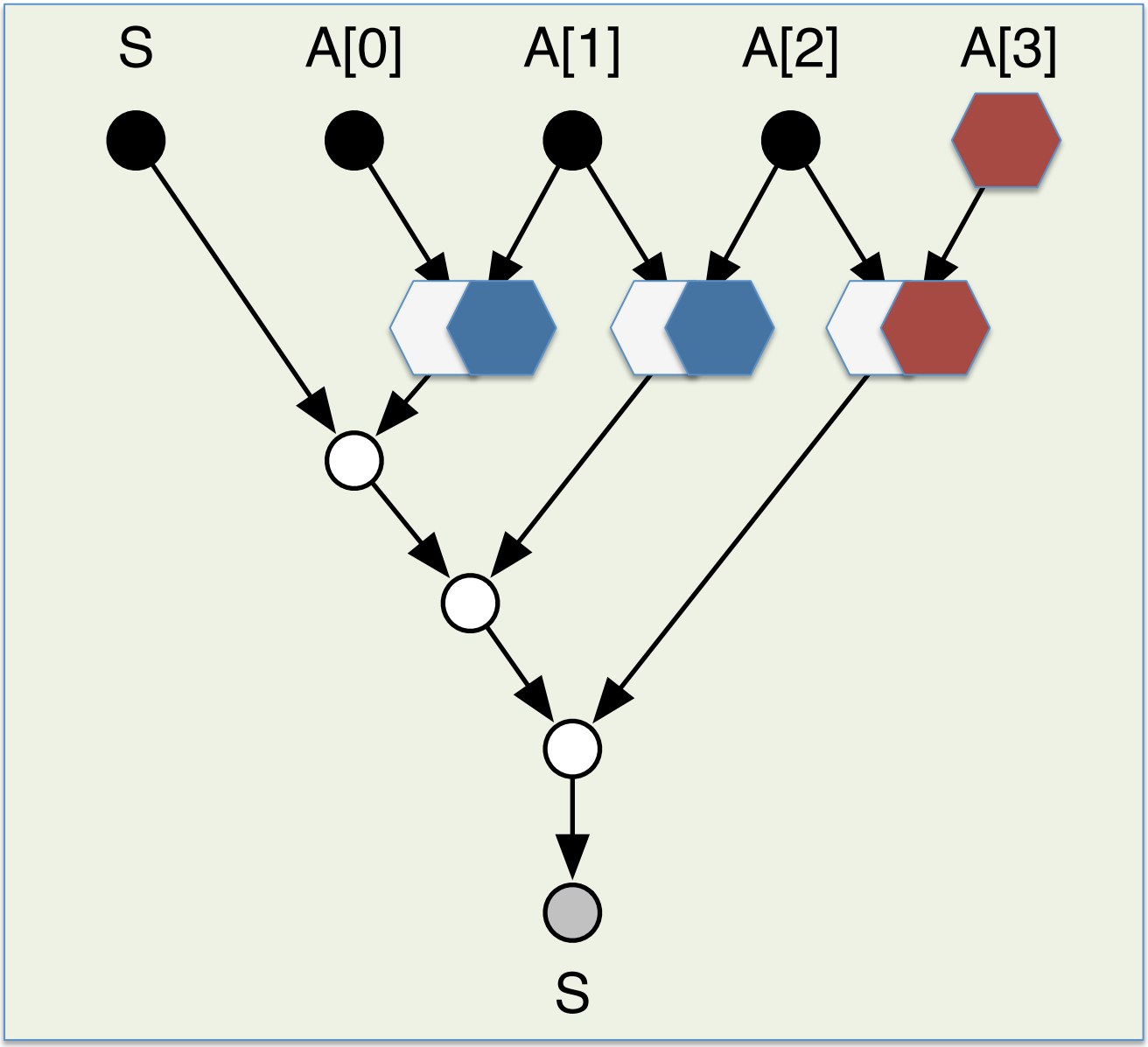
IIO RBW Pebble Game: 2 Red Pebbles



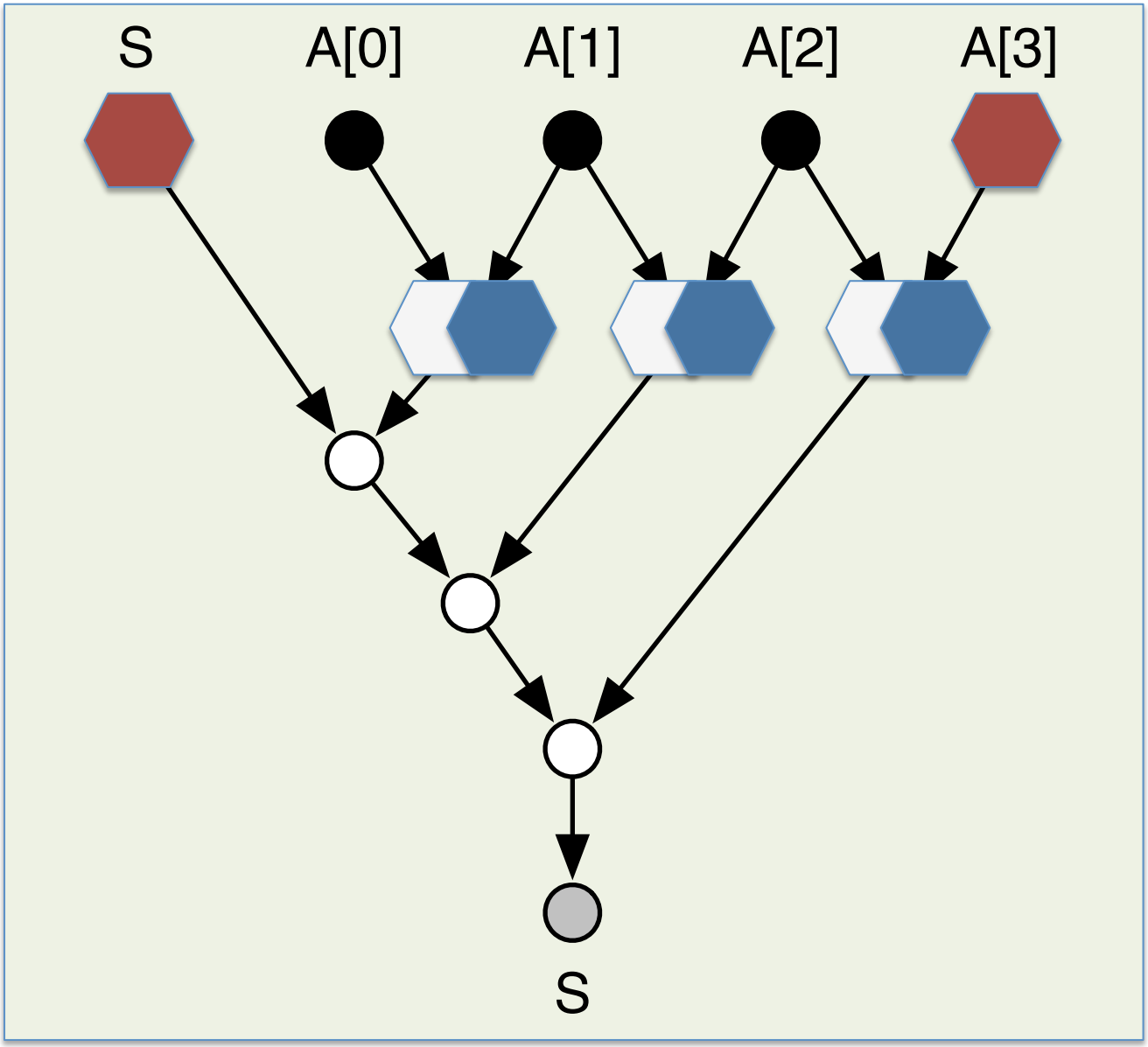
IIO RBW Pebble Game: 2 Red Pebbles



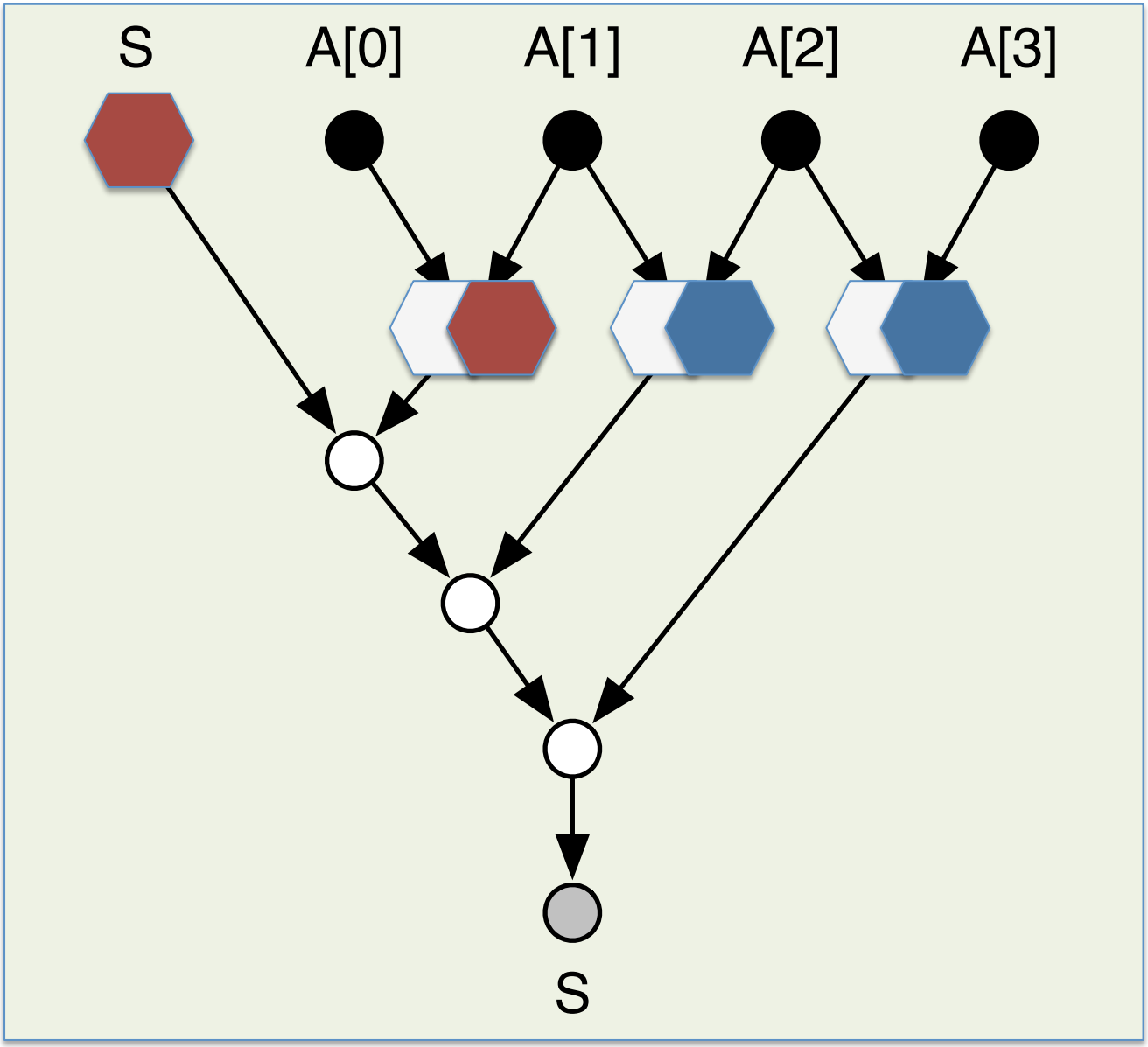
IIO RBW Pebble Game: 2 Red Pebbles



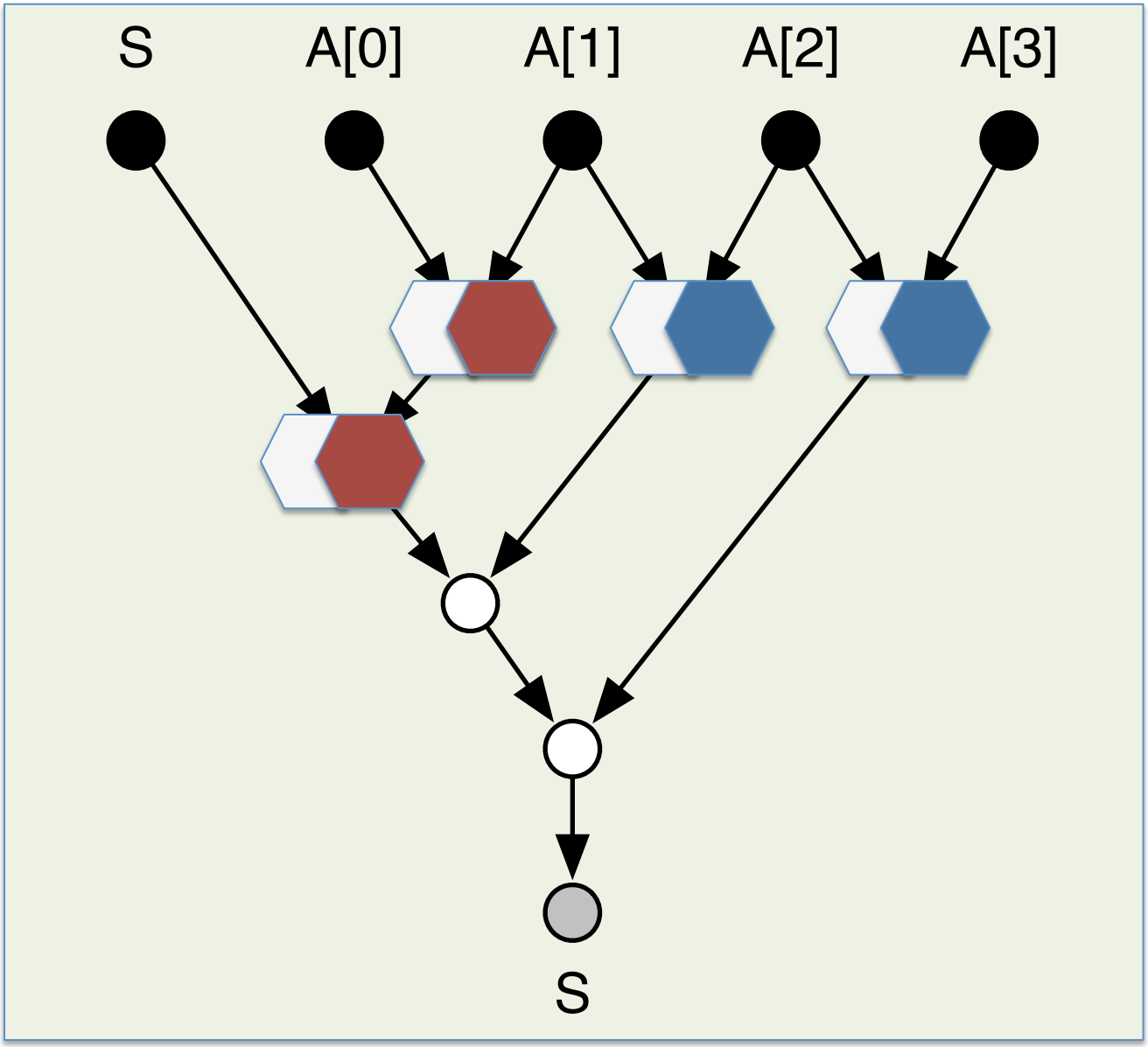
IIO RBW Pebble Game: 2 Red Pebbles



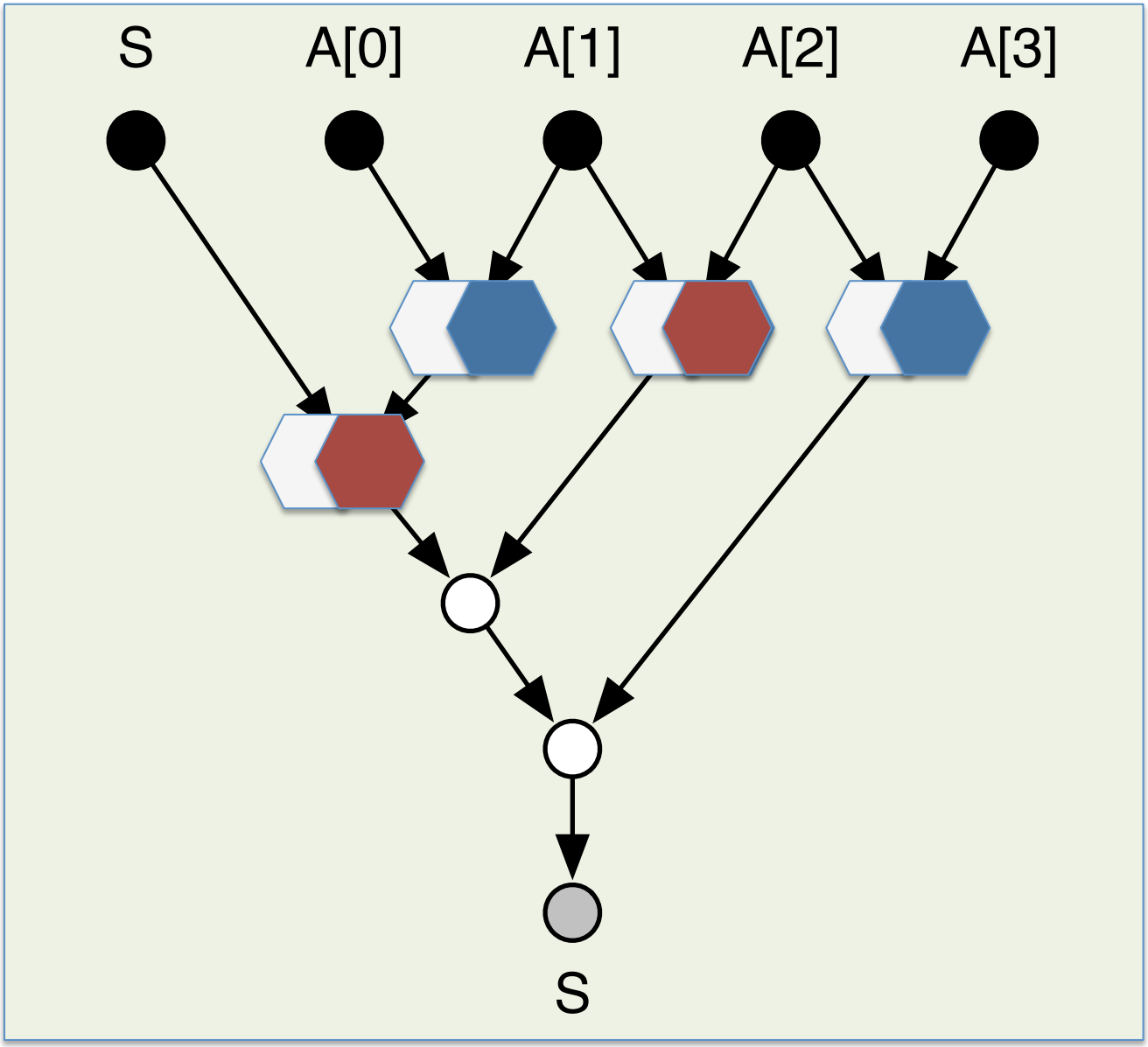
IIO RBW Pebble Game: 2 Red Pebbles



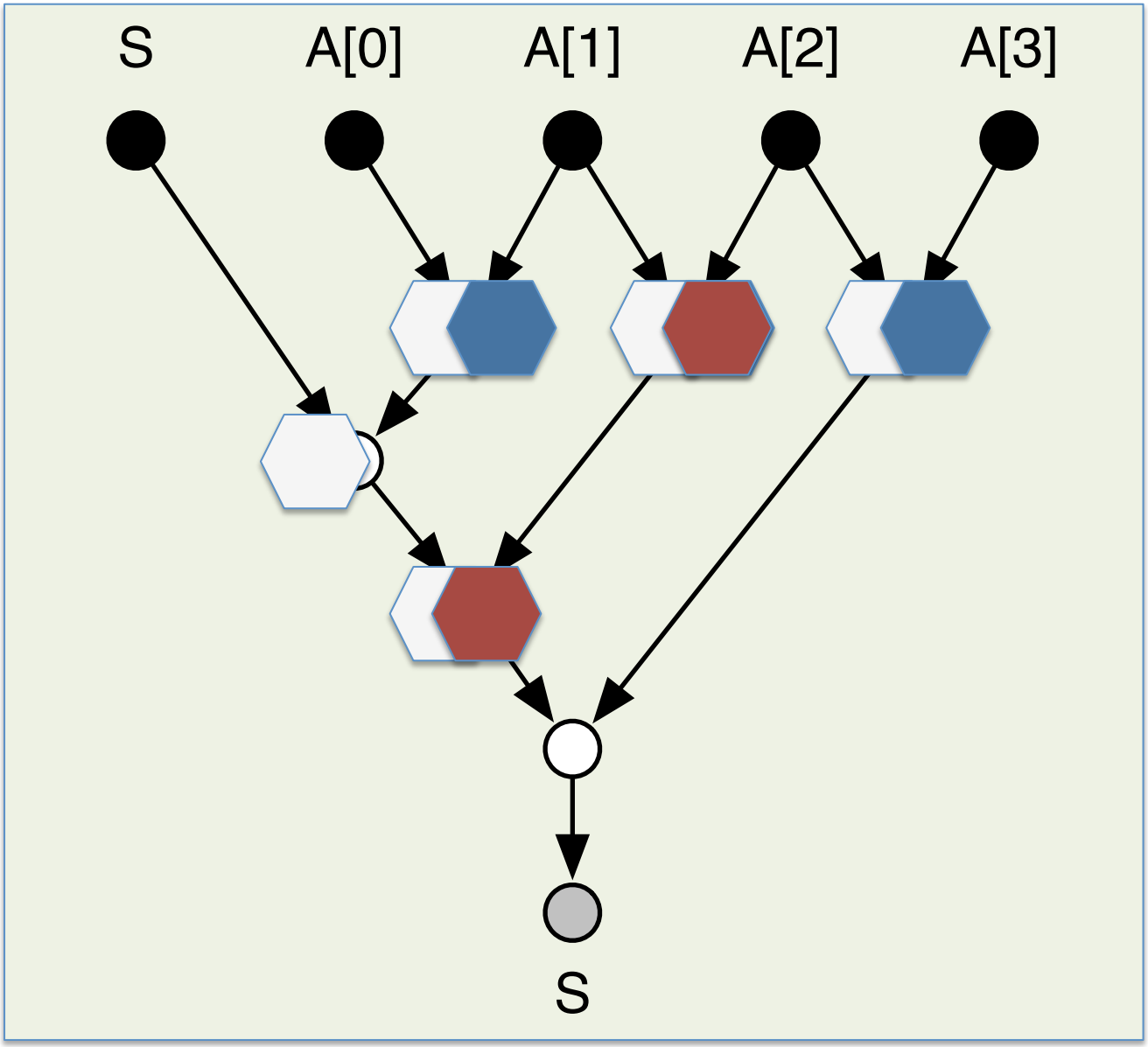
IIO RBW Pebble Game: 2 Red Pebbles



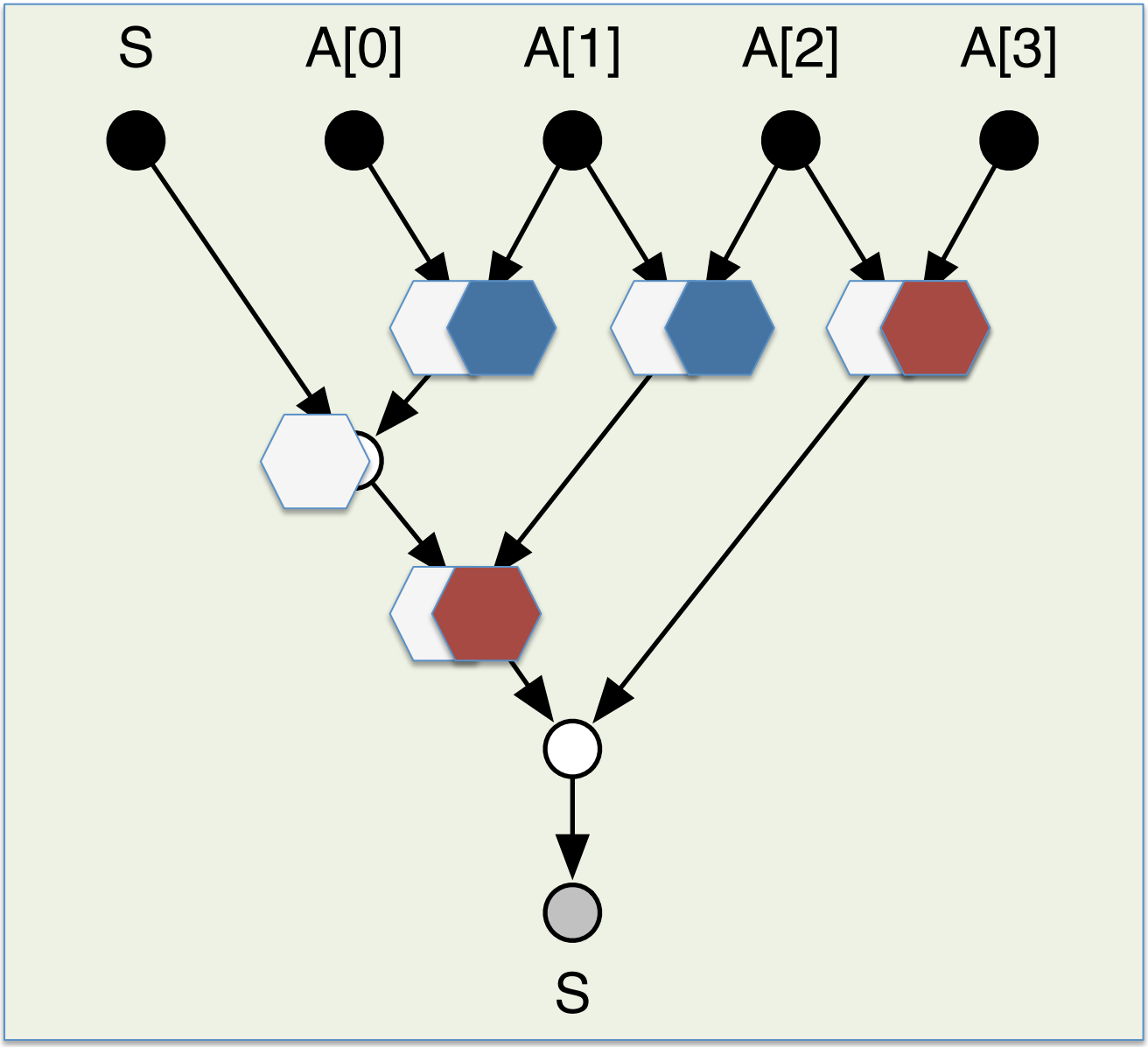
IIO RBW Pebble Game: 2 Red Pebbles



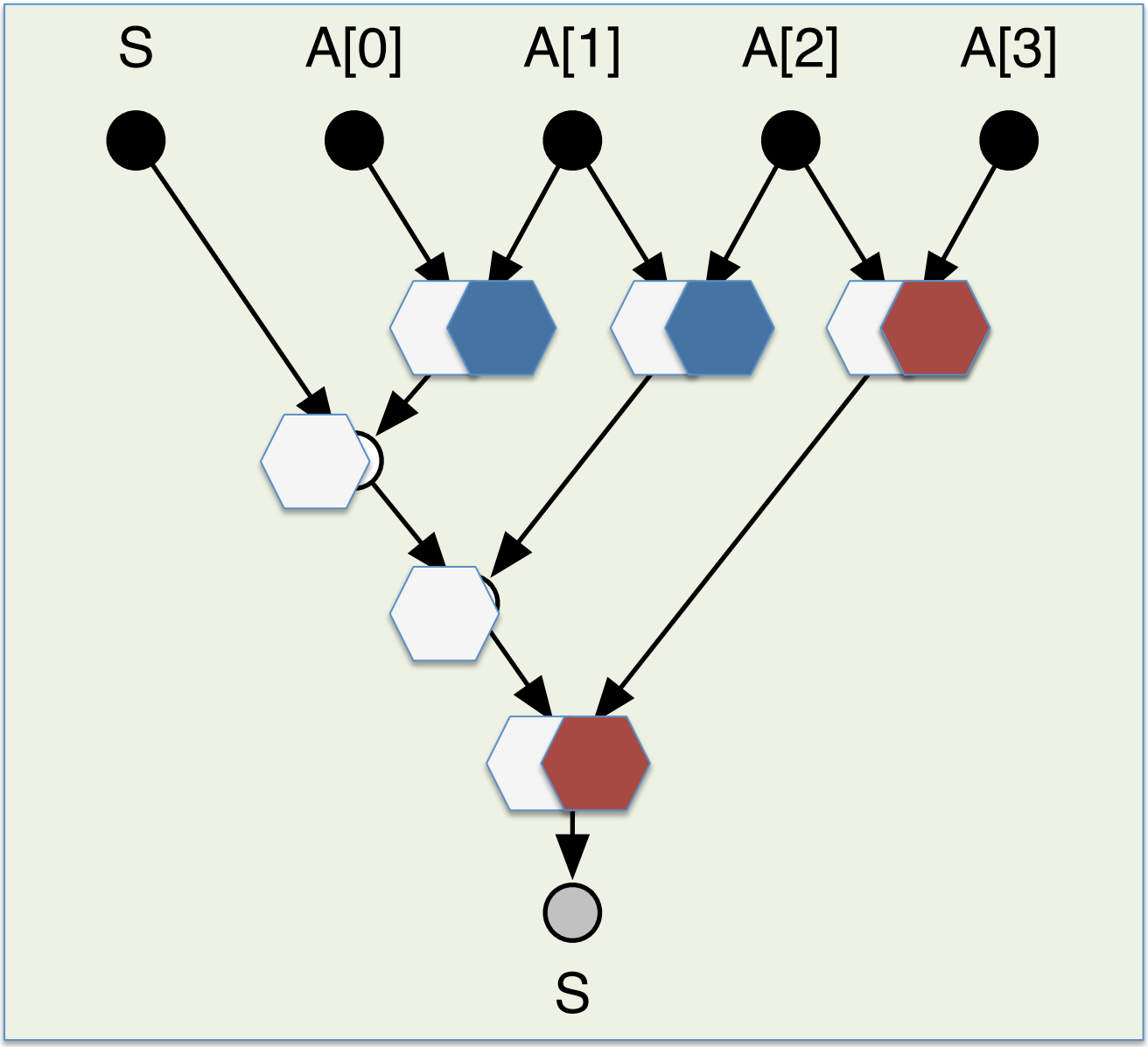
IIO RBW Pebble Game: 2 Red Pebbles



IIO RBW Pebble Game: 2 Red Pebbles

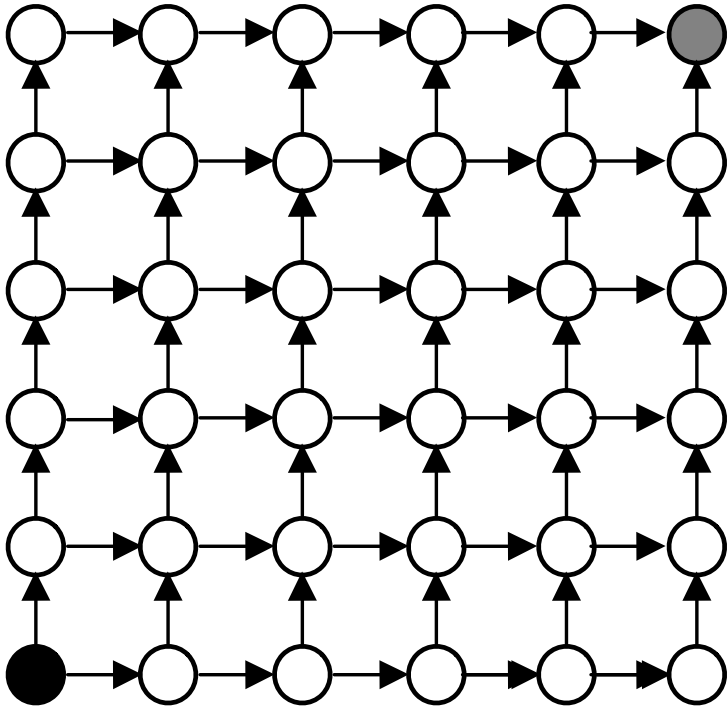


IIO RBW Pebble Game: 2 Red Pebbles



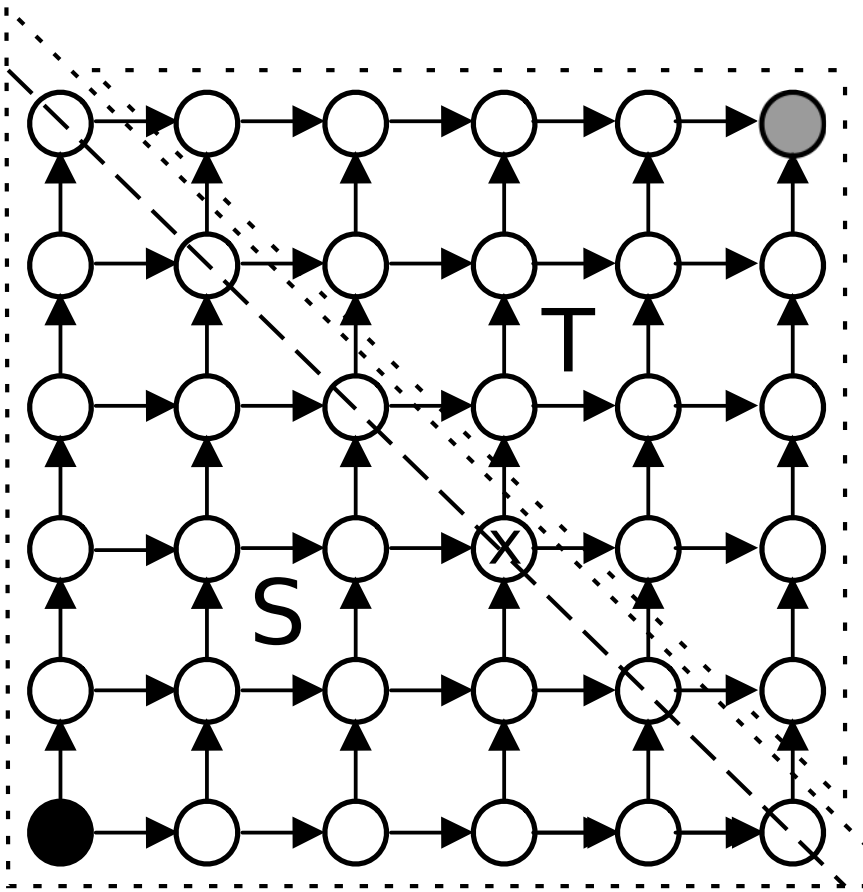
Limitation of Hong-Kung S-partition Model

- Hong/Kung's S-partitioning approach constrains vertex partitions based on size of input dominator sets
  - Not effective for computations with few inputs that generate many more intermediates and finally output few values; extreme example: Diamond DAG
  - For S>0 red pebbles, a valid 2S-partition is the entire CDAG, i.e. lower bound = S*(1-1) = 0

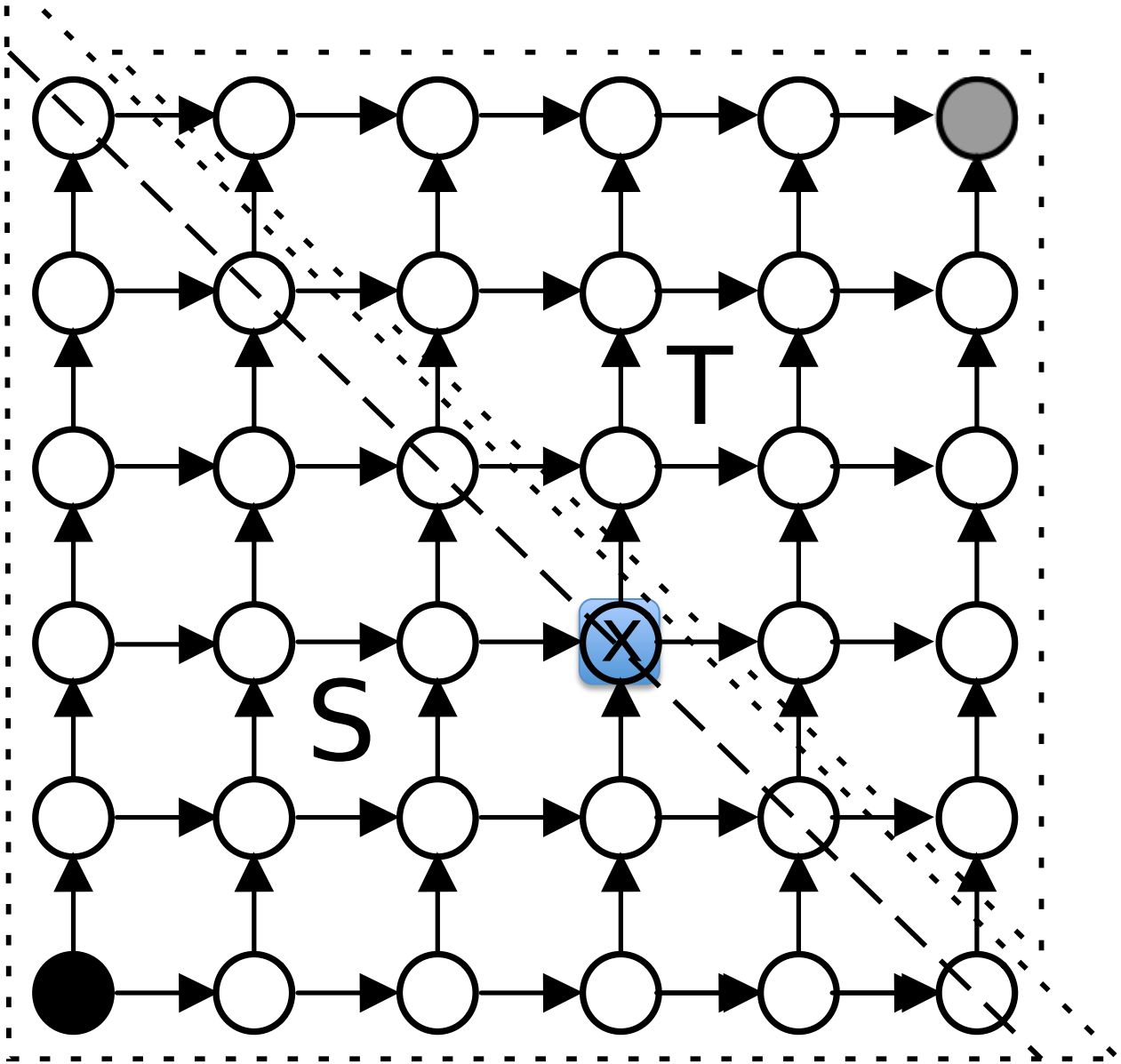


New Approach: MinCut Partitioning

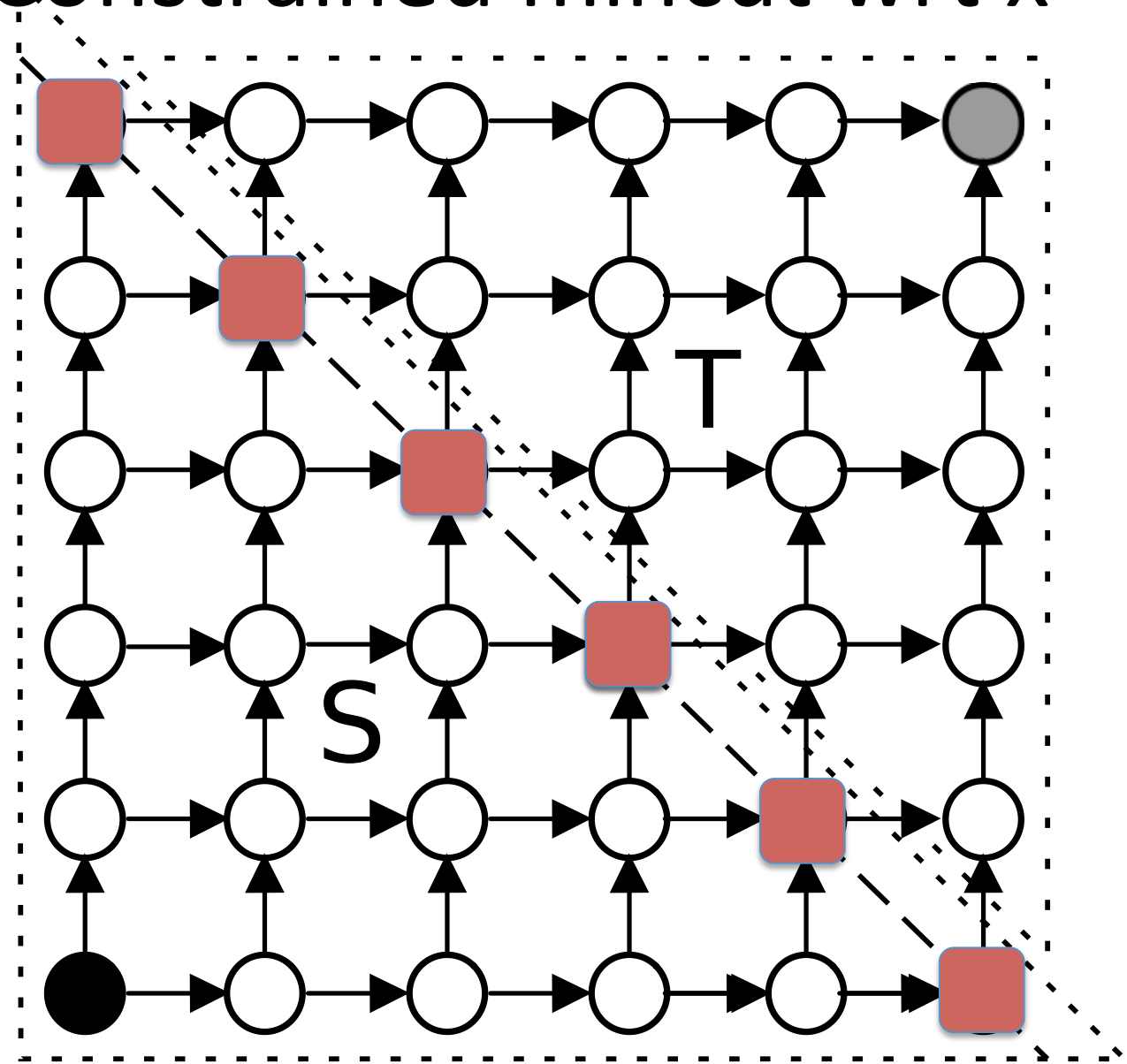
- Association between *wavefront* of "live" vertices just before firing vertex x in any valid RBW pebble game and a convex graph mincut including vertex x
- Number of vertices in maximal convex graph-mincut associated with all vertices is a lower bound on the maximal *schedule* wavefront in optimal pebble game.



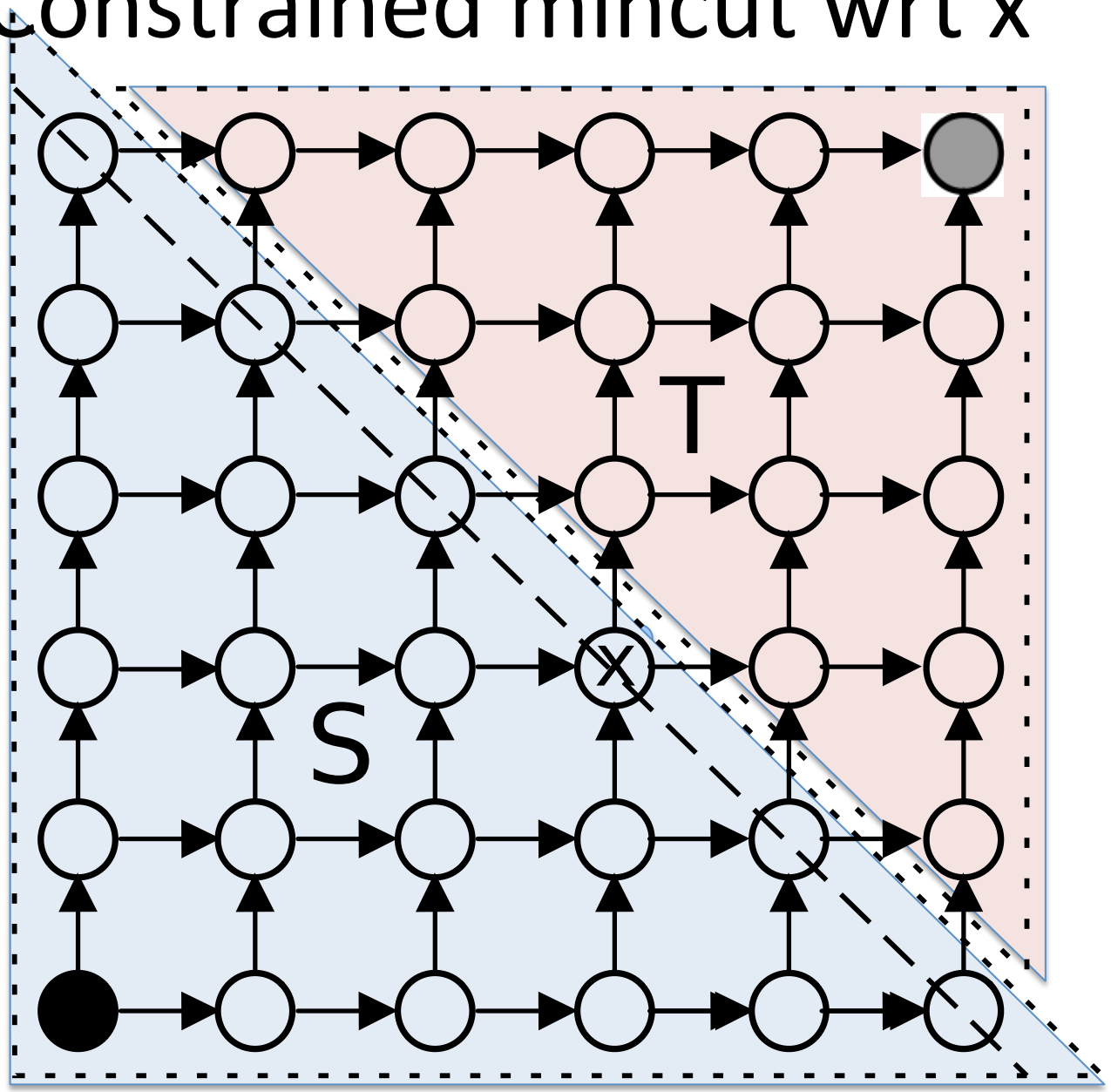
Diamond DAG with const. mincut wrt x



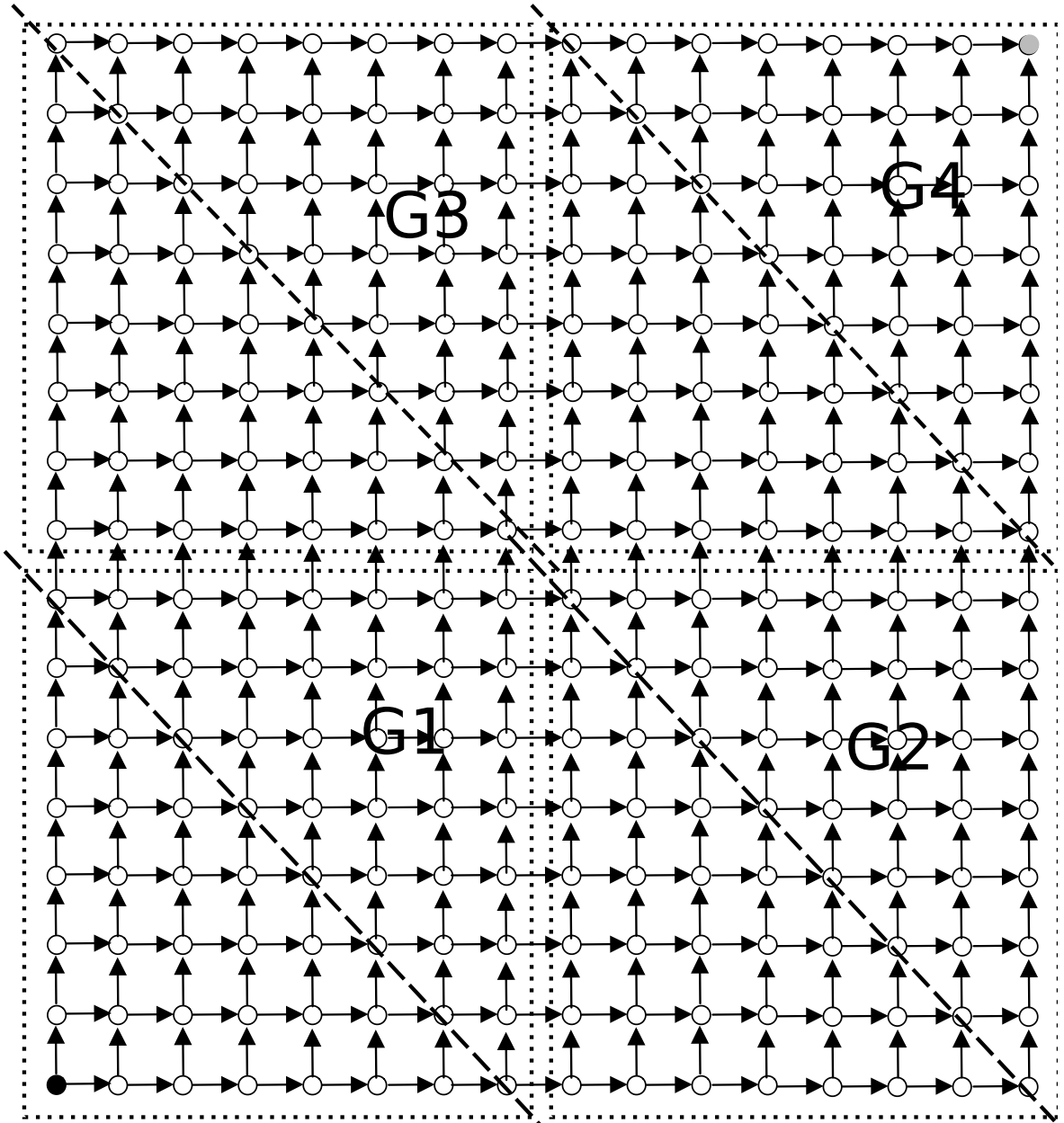
Constrained mincut wrt x



Constrained mincut wrt x



Diamond DAG with 4 disjoint vertex sets



IO Complexity Comparison

	Hong-Kung	Our bounds
Diamond DAG	0	$\frac{(n-2S)^2}{S}$
FFT	$\frac{\Omega(n\log(n))}{\log(S)}$	$\frac{2n\log(n)}{\log(S)}$
9-pt Seidel	$\Omega(\frac{n^2}{S})$	$0.75\frac{n^2}{\sqrt{S}}$

IO Lower Bounds for Arbitrary CDAGs

- Automation: Tool to generate IO lower bound for an arbitrary CDAG instance, for given number of red pebbles
- Combine both approaches and use tighter bound:
  - *2S partitioning of Hong-Kung*
  - *New convex mincut*
- Each bounding approach is effective for different class of CDAGs
  - 2S partitioning: high-fanout DAGs like Mat-Mult
  - Convex mincut: bounded fanout CDAGs like stencils

```
Function CalculateLB(G, do_mincut, do_2Kpart)
    if do_mincut then
        Q_mincut = MinCut(G);
        if Q_mincut == 0 then
            do_mincut = false;
        end
    else
        Q_mincut = 0;
    end
    if do_2KPart then
        Q_2KPart = 2KPart(G);
        if Q_2KPart == 0 then
            do_2Kpart = false;
        end
    else
        Q_2KPart = 0;
    end
    if do_mincut ∨ do_2Kpart then
        (G_1, G_2) = Bisect(G);
        Q_1 = CalculateLB(G_1, do_mincut, do_2Kpart);
        Q_2 = CalculateLB(G_2, do_mincut, do_2Kpart);
        return max(Q_mincut, Q_2KPart, Q_1, Q_2);
    else
        return 0;
    end
end
```

Summary

- Characterizing lower bounds on inherent data access complexity of algorithms is important
- Previous approaches have some limitations
- New approach to deriving lower bounds
  - Develop tighter analytical bounds for some algorithms
  - Enable composing lower bounds for constituent sub-computations
  - Generate lower bounds for irregular CDAGs